Vorlesungsskript

Wintersemester 2016

Einführung in die Theoretische Informatik

Formale Sprachen, Automatentheorie, Berechenbarkeit

Prof. Dr. Karsten Morisse

Auszug aus der Version vom 11. Dezember 2016

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Inhaltsverzeichnis

Teil I.

Einführung in die theoretische Informatik

Vorwort

Über dieses Manuskript

Dieses Vorlesungsskript ist im Rahmen der Vorlesung *Theoretische Informatik* entstanden, die ich erstmalig im Sommersemester 2010 an der Hochschule Osnabrück im Rahmen der Informatik-Studiengänge gelesen habe. An einigen Stellen ist es noch nicht ganz vollständig ausformuliert. Dies wird in folgenden Semestern behoben werden. Ziel dieses Dokumentes ist die Unterstützung der Studierenden, um sie von der Mitschrift während der Vorlesungstermine zu befreien. In folgenden Semestern wird dieses Dokument noch um das Thema Komplexitätstheorie ergänzt.

Insgesamt ist diese Ausarbeitung zur Vorlesung, wie auch die Vorlesung selbst, bewusst sehr formal und kompakt gehalten. Zum einen ist dies dem Umstand geschuldet, dass man einen Sachverhalt mit einer kurzen Formel häufig sehr viel prägnanter und exakter beschreiben kann, als eine eher umgangssprachliche Beschreibung. Zum anderen muss die Verwendung von Formelwerk durch Informatik-Studierende aber auch explizit erlernt werden. Diese Veranstaltung ist ein Versuch, dazu einen Beitrag zu leisten.

Nach einer kurzen Einleitung in Kapitel 1 werden in Kapitel 2 einige mathematischen Grundlagen wiederholt. In der entsprechenden Vorlesung werden diese Dinge nicht noch einmal wiederholt, sondern sie werden als bekannt vorausgesetzt. Das Kapitel dient daher primär zum Nachschlagen zwischenzeitlich vergessener Sachverhalte.

Hinweise:

- Dieses Vorlesungsskript ist aktuell (11. Dezember 2016) noch nicht ganz fehlerfrei und auch noch nicht ganz vollständig. Ich bin fortlaufend dabei die Fehler zu eliminieren und es zu vervollständigen.
- An einigen Stellen finden sich Links auf Youtube-Videos. Diese Videos sind kurze Screencapturing-Aufnahmen, um den jeweiligen Sachverhalt - beispielsweise ein Beweis zu einem Satz - zu erläutern. Diese Videos sind als Randnotiz in Form eines QR-Codes im Dokument zu finden. In der elektronischen Fassung dieses Dokumentes, sind diese QR-Codes anklickbare Grafiken, die auf das dahinterliegende Video verweisen. In der gedruckten Fassung kann mit einem QR-Code-Scanner der entsprechende Link ausgelesen werden. Eine vollständige Liste aller Videos befindet sich im Anhang ab Seite 369.

- Neben den YouTube-Videos gibt es eine Toolbox zum Arbeiten und Experimentieren mit einigen Konzepten der theoretischen Informatik. Die ToolboxTI ist eine interaktive Software-Anwendung, mit deren Hilfe man mit Grammatiken und Automaten arbeiten kann. In der elektronischen Fassung dieses Scriptes ist die ToolboxTI-Randnotiz anklickbar und man wird auf ein entsprechendes Beispiel in der Anwendung geführt. Diese Toolbox befindet sich noch im Aufbau und wird schrittweise weiter ausgebaut. Sie ist erreichbar unter der URL:

ToolboxTI

https://toolboxti.hs-osnabrueck.de

Inhaltliche Vorgehensweise und Lehrveranstaltungskonzept

Bei der Durchsicht verschiedener Bücher zur Theoretischen Informatik können unterschiedliche Herangehensweisen beobachtet werden. In aller Regel findet man eine Trennung in Formale Sprachen, Automatentheorie, Berechenbarkeitstheorie und Komplexitätstheorie. Die dabei vorgenommene Reihenfolge variiert jedoch. Während eines der Standardwerke der Einführung zur Theoretischen Informatik [HMU11] von den einfachen Automatenmodellen (endlichen Automaten) zu den mächtigeren Konzepten der Automatentheorie übergeht und erst daran die Überlegungen zur Berechenbarkeit und Komplexität anschliesst (diesem Vorgehen wird z.B. auch in [LP98] oder [Hof09] gefolgt), geht [Weg05] den umgekehrten Weg, indem er zunächst auf Basis der Turing-Maschinen die Begriffe der Entscheidbarkeit darstellt und die Komplexitätstheorie sowie im Anschluss daran die übrigen Sprachklassen der Chomsky-Hierarchie behandelt.

In diesem Werk und der zugehörigen Lehrveranstaltung werden nach und nach die einzelnen Sprachklassen der Chomsky-Hierarchie behandelt. Dabei bewegen wir uns - Mengentheoretisch betrachtet - von innen nach aussen, d.h. zunächst werden die Konzepte der Sprachklasse der regulären oder rationalen Sprachen behandelt. Danach werden schrittweise (oder kapitelweise) die mächtigeren Sprachklassen behandelt. Dabei werden jeweils die Konzepte zur Erzeugung einer Sprache (Grammatiken) und die Modelle zur Verarbeitung einer Sprache (Automaten) behandelt. Eine der zentralen Fragestellungen bildet dabei die Frage nach der Lösbarkeit des Wortproblems für die jeweilige Sprachklasse. Nach meiner Auffassung erleichtert dies den Zugang zu den Ideen und Konzepten der theoretischen Informatik. Endliche Automaten sind ein sehr überschaubares Konzept und die aufgeworfenen Fragen lassen sich in den meisten Fällen konstruktiv lösen. Insofern wird die Theoretische Informatik an dieser Stelle sehr praktisch und anschaulich. Dennoch wird das Fach von vielen Studierenden als schwierig empfunden. Der erforderliche Formalismus wirkt abschreckend und die Prüfungsergebnisse, sofern die Studierenden die Prüfung nicht immer wieder aufschieben, sind für die Lehrenden häufig ernüchternd. Ein vergleichsweise hoher Anteil der Studierenden muss die Prüfung wiederholen. Um dem entgegenzuwirken, ist das vorliegende Script die Grundlage für ein andersartiges Veranstaltungskonzept.

Dieses Werk bildet die Grundlage für die Einführungsvorlesung Theoretische Informatik, die an der Hochschule Osnabrück als Pflichtveranstaltung mit 4 Semesterwochenstunden im 4. Semester der Informatik-Studiengänge gelehrt wird. Sie wird nach dem Konzept des Inverted oder Flipped Classroom ([BS12] oder [Mor15b]) gelehrt. Da dieses Konzept in den Hochschulen erst zögernd Einzug hält, soll es an dieser Stelle kurz erläutert werden (siehe auch Abbildung 0.1). Beim ICM sind die Präsenzphase, d.h. die gemeinsame Zeit von Studierenden und Lehrenden und die individuelle Lernphase im Vergleich zur traditionellen Lehre in der Abfolge vertauscht. Anstelle der Wissensaneignung als Vorlesung in Präsenz tritt eine Wissensaneignung individuell durch die Studierenden. Die gemeinsame Zeit von Studierenden und Lehrenden wird hingegen genutzt, um das individuell erworbene Wissen durch praktische Anwendungen und Übungen zu vertiefen und zu verfestigen. In dieser wichtigen Phase steht der Lehrende als Berater, Begleiter, Coach zur Verfügung. Zu den Zeitpunkten, wo den Studierenden deutlich wird, dass sie den ein oder anderen Aspekt vielleicht doch nicht verstanden haben (das fällt ja häufig bei der Bearbeitung von Übungsaufgaben auf), sind sie beim traditionellen Lehrkonzept auf sich alleine gestellt bzw. auf eine Lerngruppe angewiesen. Beim ICM hingegen steht der oder die Lehrende in dieser wichtigen Phase als Ansprechperson zur Verfügung. Das erfordert vom Lehrenden allerdings das eigene Rollenverständnis neu zu definieren. Er oder sie nimmt jetzt nicht mehr die Rolle der Wissensvermittlung ein, sondern ist vielmehr ein Coach als Begleitung der studentischen Lernprozesse.

Um eine Lehrveranstaltung in dieser Form durchzuführen, bedarf es zwischen den Teilneh-

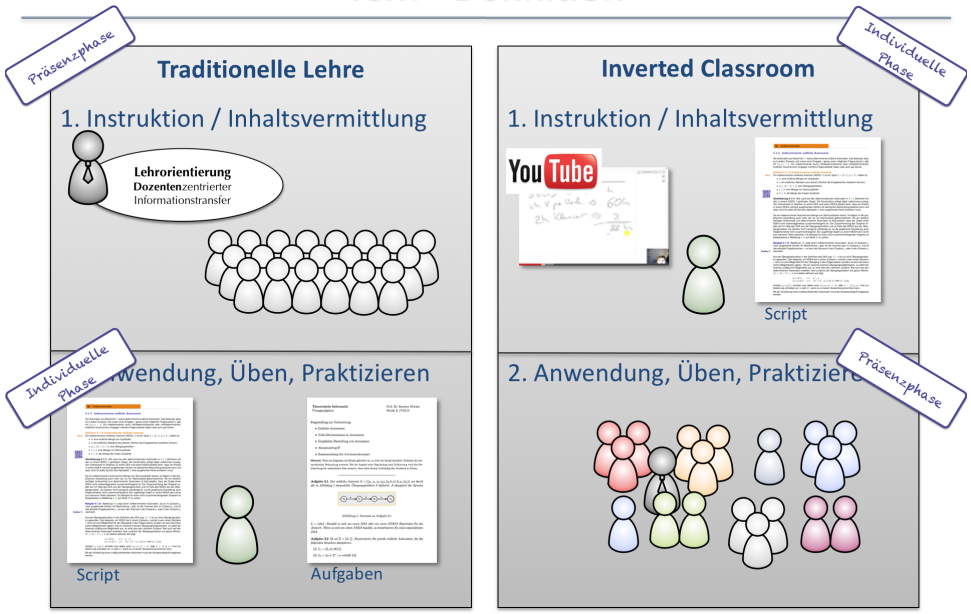


Abbildung 0.1.: Lehrkonzept ICM: Präsenzphase und individuelle Phase

mern der Veranstaltung einige Vereinbarungen:

- Jede Woche gibt es zwei Veranstaltungstermine mit Studierenden und dem Lehrenden. Anstatt jedoch den Lehrinhalt in Form von PowerPoint-Folien oder an der Tafel zu präsentieren, wird diese Zeit genutzt, um die individuellen Lernprozesse der Studierenden zu unterstützen.
- Dies erfordert seitens der Studierenden eine explizite Vorbereitung dieser Präsenztermine. Dazu wird vom Dozenten wöchentlich ein Themenbereich festgelegt, auf den sich die Studierenden selbstständig mit Hilfe dieses Scriptes, der entsprechenden Videos oder weiterer Literatur vorbereiten.
- Vor der selbstständigen Erarbeitung werden neue Themenbereiche einleitend kurz vorgestellt. Beispielweise in 5 - 10 Minuten die Arbeitsweise eines Push-Down-Automaten verglichen zu einem endlichen Automaten (wenn dieser in vorhergehenden Stunden bereits behandelt wurde).
- Die selbstständig erarbeiteten Themenbereiche werden im Rahmen der Präsenztermine nicht noch einmal präsentiert oder *vorgelesen*. In den gemeinsamen Veranstaltungsterminen werden lediglich offen gebliebene Fragen diskutiert. In Ausnahmefällen (z.B. bei einem erkennbaren breiten Unverständnis) werden einzelne Aspekte des Stoffes noch einmal dargestellt.
- Zur Verfestigung des erarbeiteten Materials gibt es Übungsaufgaben zu den jeweiligen Themen. Diese Aufgaben werden in Kleingruppen ($2-4$ Studierende) bearbeitet. Hierbei wird Wert darauf gelegt, das die Studierenden die Aufgaben im Team bearbeiten und gegenseitig voneinander lernen können.
- Zudem werden die Präsenztermine durch verschiedene Interaktionsübungen (Clicker, 3×3, Aktives Plenum, ...) belebt.

Diese Vorgehensweise soll die Rolle der Studierenden ändern (siehe Abbildung 0.2). Die in der klassischen Vorlesung häufig eingenommene Rolle des passiven Wissensempfänger wird

aufgelöst zugunsten der Rolle eines aktiven Lerners und im Idealfall eines unabhängig Lernenden. Damit dieses Veranstaltungsformat erfolgreich ist, bedarf es seitens aller Teilnehmer einer ernsthaften Vorbereitung der einzelnen Veranstaltungstermine. Dies kann durch die Studierenden einzeln erfolgen, aber es ist natürlich auch die Vorbereitung in Lerngruppen denkbar.

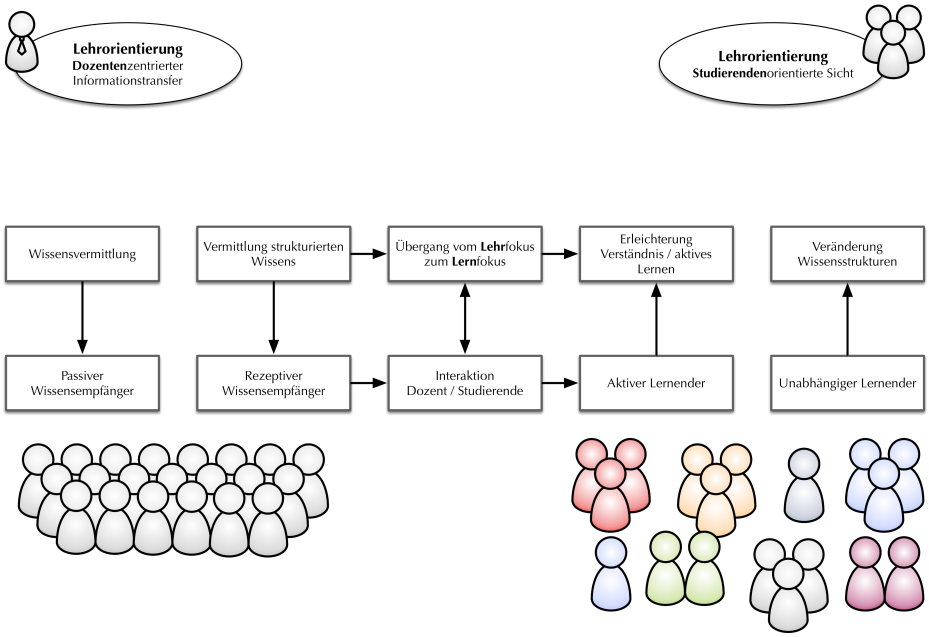


Abbildung 0.2.: Lehrkonzept: Von der dozentenzentrierten Perspektive zur Studierendenorientierten Sicht (in Anlehnung an [Win08])

Kapitel 1.

Einführung

1.1. Was ist Informatik?

> *Informatik ist die Wissenschaft von der systematischen Verarbeitung von Informationen, besonders der automatischen Verarbeitung mit Hilfe von Digitalrechnern.*

In dieser Definition aus [(Hr93] steckt im Kern die automatisierte Verarbeitung von Informationen, die sich heute in einer sehr vielfältigen Ausprägung von Anwendungen wieder findet. Welcher Wissenschaftsdisziplin kann die Informatik aber zugeordnet werden?

Ist die Informatik eine Naturwissenschaft?

Naturwissenschaft

Naturwissenschaften sind beispielsweise die Bereiche der Astronomie, Physik, Chemie oder Biologie. Hierbei geht es um die Erforschung und Ergründung der Natur und der darin ablaufenden Prozesse. Dabei werden die Abläufe der Natur beobachtet, gemessen und analysiert. Einerseits wird versucht, eine dabei gegebenenfalls auftretende Regelmäßigkeit zu erkennen. Neben dem reinen Verständnis der in der Natur vorkommenden Abläufe ist es aber auch eine wichtige Aufgabe geworden, die Natur nutzbar zu machen. Insofern bilden die Naturwissenschaften ein wichtiges theoretisches Fundament für Technik, Medizin oder Umweltthemen. Auch die Informatik enthält Methodiken und Aspekte einer Naturwissenschaft. In der Informatik wird mit Informationsobjekten und deren Verarbeitung durch Algorithmen, Programme und Rechner gearbeitet. Die physikalisch existierenden Prozesse der Informationsverarbeitung entsprechen in diesem Sinne den in der Natur ablaufenden Prozessen und die Informationsobjekte sind das Pendant zu den Objekten der Natur. Auch wird in Teilgebieten der Informatik versucht die Erkenntnisse der Natur einfliessen zu lassen, beispielsweise in der Algorithmenentwicklung bei Genetische Algorithmen.

Ingenieurwissenschaft

Für viele Informatiker ist die Informatik eine eher anwendungs- und problemorientierte **Ingenieurwissenschaft**. Die Ingenieurwissenschaften sind angewandte Wissenschaften, die sich mit den technischen Entwicklungen und der Konstruktion von Verfahren, Methoden, Anwendungen und Geräten beschäftigt. Dabei geht es um den praktischen Einsatz bzw. die Umsetzung naturwissenschaftlicher Erkenntnisse bei der Erstellung technischer Produkte oder Prozesse. Die Informatik ist einerseits Grundlagendisziplin für die Ingenieurwissenschaften, liefert beispielsweise mit der Möglichkeit zur Durchführung von Simulationen ein wichtiges Hilfsmittel und Instrument für die Ingenieurwissenschaften. Andererseits sind Teile der Informatik selbst aber auch eine anwendungs- und problemorientierte Disziplin und besitzt daher Gemeinsamkeiten mit den Ingenieurwissenschaften, beispielsweise im Bereich des Software-Engineering.

Es gibt aber auch wichtige Bezüge zu den Sozial- oder Gesellschaftswissenschaften, in denen Aspekte des gesellschaftlichen Zusammenlebens der Menschen untersucht werden. Der Wissenschaftler PETER KRUSE spricht beispielsweise von der Revolution im Internet durch kollektive Bewegungen.

Formalwissenschaft

Die theoretische Informatik ist ein Teilgebiet der Informatik, die als Formalwissenschaft bezeichnet wird: es geht um die Betrachtung und Analyse von Formalen Systemen. Aber die

Informatik ist mehr. Wie auch die Philosophie und Mathematik sind in der Informatik auch allgemeine Begriffe Gegenstand der Betrachtung

> Information, Komplexität, Berechenbarkeit, Sprache, Determinismus, Beweis, Wissen, Kommunikation, Algorithmus, Zufall, ...

und tragen zum Verständnis bei. Einigen dieser Begriffe hat die Informatik eine neue oder zusätzliche Bedeutung gegeben.

1.2. Was ist theoretische Informatik?

Die Theoretische Informatik ist neben der *Praktischen Informatik*, der *Technischen Informatik* und der *Angewandten Informatik* einer der zentralen Bereiche der Informatik. In der theoretischen Informatik beschäftigt man sich mit abstrakten Konzepten, formalen Systemen, Modellen und Methoden, die sich hinter modernen Computersystemen verbergen. Dabei befreit man sich aber von dem konkreten technischen Aufbau eines Computersystems, sondern fokussiert auf den zugrundeliegenden Kern und die Ideen hinter einem Computer. Die Theoretische Informatik erweckt dadurch den Anschein einer gewissen Praxisferne oder geringen Anwendungsnähe. Der Ursprung der Theoretischen Informatik liegt in der Mathematik. Inhaltlich befasst sich die Theoretische Informatik mit den Bereichen Formale Sprachen, Automatentheorie, Berechenbarkeitstheorie, Komplexitätstheorie und der Logik. Auf diese Bereiche gehen wir im folgenden kurz ein.

Formale Sprachen Betrachten wir den folgenden Programmtext

```
1  int fun (int[] x) { int s = x[0]; for (int i=0; i<x.length;i++)
2          if (x[i] > s) s = x[i]; return s; }
```

Wir können das Verständnis für diesen Text erhöhen, indem wir die Lesbarkeit vereinfachen und beispielsweise Einrückungen und unterschiedliche Farben für Schlüsselworte und Bezeichner verwenden.

```
1  int fun (int[] x) {
2          int s = x[0];
3          for (int i=0; i<x.length;i++)
4                  if (x[i] > s)
5                          s = x[i];
6          return s;
7          }
```

Ohne diese visuellen Unterstützung fällt das Verständnis schon deutlich schwerer. Wie aber kann ein Computer einen solchen Text auswerten? Wie erkennt und interpretiert der Computer den Text und leitet Handlungsanweisungen daraus für ihn ab? Mit dem Verständnis des Textes muss man gar nicht erst anfangen. Zunächst geht es einmal darum, die Syntax des Textes zu erkennen und zu bewerten. Der Computer muss in die Lage versetzt werden, einen gegebenen Programmtext zu analysieren. Man spricht dabei auch vom *Parsen* des Textes. Gleiches gilt bei der Auswertung und Darstellung einer HTML-Seite durch einen Browser, bei der Überprüfung einer Benutzereingabe oder bei der Prüfung einer Nachricht auf eine bestimmtes Kommunikationsprotokoll.

Parsen

Es geht also um die Grundlagen von Programmiersprachen und deren Analyse und Übersetzung in ausführbaren Programmcode. Eine Programmiersprache wird durch ein System von Regeln definiert, die als *Grammatik* bezeichnet werden. Dabei interessiert die Frage, welche Regeltypen erforderlich sind beziehungsweise wie mächtig müssen die

Grammatik

Wortproblem

Regeln der Grammatik sein, um eine Programmiersprache zu beschreiben. Eine wichtige Fragestellung in diesem Zusammenhang spielt das *Wortproblem*. Das Wortproblem ist die Frage, ob ein Wort w zu einer bestimmten Sprache L gehört oder nicht? Lässt sich ein Wort mit den Regeln einer Grammatik erzeugen oder nicht und wie effizient lässt sich dieser Nachweis führen? Im Kontext der Programmiersprachen ist es die Frage: ist ein gegebenes Programm ein gültiges *Wort* aus der Sprache aller gültigen Programme der gegebenen Programmiersprache, d.h. ist die Syntax des Programmes korrekt, also entsprechend den Regeln der zugrundeliegenden Grammatik.

Die Theorie der formalen Sprachen beschäftigt sich also mit der regelbasierten Erzeugung von Wortmengen sowie der Analyse und Klassifikation von Wörtern und Sprachen. Die zugrundeliegende Theorie verschafft einem Methoden und Hilfsmittel, die für den systematischen Umgang mit modernen Programmiersprachen und dem Compilerbau erforderlich sind.

Automatentheorie In der Automatentheorie behandelt man abstrakte Maschinenmodelle, welche zur Modellierung, Analyse und Synthese zustandsbasierte Systeme verwendet werden. Einige Automatenmodelle haben einen engen Bezug zu den formalen Sprachen, da mit Ihnen Wörter aus formalen Sprachen verarbeitet werden können. Andere Modelle wie beispielsweise die *Turing-Maschine* bilden die Grundlage für die Frage nach der Berechenbarkeit eines Problems oder einer Funktion.

Turing-Maschine

Berechenbarkeitstheorie Wie könnte man den Wert eines folgenden Programmes P beziffern?

Eingabe: C-Programm Q, Spezifikation des Programms

Ausgabe: Ja, falls das Programm Q der Spezifikation entspricht; Nein andernfalls.

Das Programm P ist also in der Lage, jedes andere Programm Q dahingehend zu überprüfen, ob Q einer vorgegebenen Spezifikation entspricht. D.h. für jede mögliche Eingabe kann überprüft werden, ob die passende Ausgabe erfolgt. Man stelle sich einmal vor, wieviel Zeit und Arbeit einem Lehrer in der Programmierausbildung abgenommen würde, wenn er oder sie ein solches Programm P besitzen würden. Denn gerade in einer Informatik-Einführungsveranstaltung bestehen ja viele Aufgaben darin, dass man eine Programm-Spezifikation angibt und die Studierenden ein entsprechendes Programm entwickeln sollen. Der Zweck von P besteht also darin, die Spezifikation eines beliebigen Programmes zu überprüfen, zu verifizieren. Ein schwieriges Problem?

- Man muss ja nur die gültigen Eingaben der Reihe nach durchgehen....es könnte aber unendlich viele geben!
- Bei der Eingabe x läuft P aber lange, hört das denn niemals auf?

Ein wichtiges Ergebnis wird sein, dass wir selbst bei einem Teilproblem dieser Fragestellungen kapitulieren werden, nämlich die Frage, ob ein Programm bei einer beliebigen Eingabe nach endlicher Zeit überhaupt anhält. Selbst für dieses einfache *Halteproblem* werden wir zeigen, dass es prinzipiell nicht mit Hilfe eines Computers lösbar ist. Egal wie leistungsfähig die CPU auch sein mag. Es geht einfach nicht!

Halteproblem

Überlegungen zur grundsätzlichen algorithmischen Lösbarkeit von Problemen bildet also den Kern der Berechenbarkeitstheorie in Kapitel 8. Dabei werden die Grenzen der Berechenbarkeit, also die Beantwortung der Frage "Was ist überhaupt berechenbar?", behandelt. Als Grundlage wird uns dabei die Turing-Maschine dienen. Dieses ist aber nur ein Berechnungsmodell der Informatik. Daneben gibt es noch weitere. Gemäß der *Church'schen These* (siehe 8.3) gibt es aber kein Modell, welches grundsätzlich mächtiger ist als die Turing-Maschine. Alle Probleme, die mittels eines anderen Modells gelöst oder berechnet werden können, können auch mit dem Modell der Turing-Maschine gelöst beziehungsweise berechnet werden.

Church'schen These

Komplexitätstheorie Im letzten Teil widmen wir uns wieder lösbaren Problemen und uns beschäftigt die Frage, was aus Sicht der Informatik einfache und was schwierige Probleme sind. Es gibt viele Optimierungsprobleme, die einfach lösbar sind, da man für jede Eingabe nur unter endlich vielen Alternativen eine, nämlich die beste, auswählen muss. Gibt es allerdings beliebig (also unendlich) viele gültige Eingaben, kann dieser naive Ansatz nicht zum Ziel führen. Um in solch einer Problemkonstellation zum Ziel zu gelangen, ist man an effizienten Algorithmen interessiert. In der Komplexitätstheorie geht es um die quantitative Bewertung einer algorithmischen Lösungsstrategie für eine Problemstellung aber auch um die prinzipielle Existenz oder auch Nicht-Existenz von effizienten Algorithmen für Probleme. Dabei erfolgt eine Einteilung von Lösungsalgorithmen hinsichtlich Speicherplatzbedarf und Zeitbedarf in Komplexitätsklassen. Über die *NP-Vollständigkeit* eines Problems wird angenommen, dass prinzipiell kein effizienter Algorithmus für das Problem existiert. Auswege aus diesem Dilemma gibt es in Form von probabilistischen Algorithmen oder heuristischen Methoden.

NP-Vollständigkeit

1.3. Theoretische Informatik im Studium

Einführungsvorlesungen in die Theoretische Informatik gehören an deutschen Hochschulen zum Pflichtprogramm in Studiengängen der Informatik. Die Begeisterung seitens der Studierenden hält sich aber meistens in Grenzen. Das hat sicherlich verschiedene Gründe. Das mitunter hohe Abstraktionsniveau der theoretischen Informatik wirkt häufig abschreckend auf die Studierenden. Das liegt einerseits sicherlich an einem starr und abschreckend wirkenden Formalismus. Abschreckend aber nur deshalb, weil teilweise das mathematische Handwerkzeug bei den Studierenden nicht hinreichend verfestigt ist.

Ein zweiter Aspekt ist der auf den ersten Blick nicht sofort erkennbare Anwendungsbezug. Im Gegensatz zu vielen anderen Veranstaltungen eines Informatik-Studiums, wo es einen unmittelbaren Anwendungsbezug gibt, der in den die Vorlesung begleitenden Praktika unmittelbar hergestellt wird, bestehen Vorlesungen der Theoretischen Informatik in den meisten Fällen aus Vorlesungen, gegebenenfalls noch mit begleitenden Übungen. Demgegenüber steht aber der Erkenntnisgewinn, der mit den Ergebnissen der Theoretischen Informatik einhergeht. Die Ergebnisse sind häufig allgemeiner und weitreichender als in anderen Bereichen der Informatik. Sie wirken nicht immer unmittelbar aber doch mittelbar auf weitere Bereiche der Informatik ein.

So sind die Folgerungen aus dem Halteproblem (siehe 8.5.7) oder dem Satz von Rice (siehe Abschnitt 8.6) auch für die praktischen Einsatzmöglichkeiten der Informatik insgesamt sehr weitreichend. Eine gerne gestellte Prüfungsfrage ist die Frage nach einem *Sicherungsnetz* im Bereich der Software-Entwicklung: gibt es ein allmächtiges Kontrollprogramm K, welches als Eingabe ein in einer beliebigen Programmiersprache entwickeltes Programm P sowie die zugehörige formale Spezifikation $S(P)$ nimmt und die Frage beantwortet, ob P sich korrekt entsprechend der Spezifikation $S(P)$ verhält. Immer. Bei allen möglichen Eingaben. Etwas formaler ausgedrückt:

$$K(P, S(P)) = \begin{cases} 1 & : \quad P \text{ genügt der Spezifikation } S(P) \\ 0 & : \quad P \text{ genügt der Spezifikation } S(P) \text{ nicht} \end{cases}$$

Es ist die Konsequenz aus dem Satz von Rice, dass es ein solches Verifikationsprogramm K leider nicht gibt. Zumindest nicht in aller Allgemeingültigkeit. Alleine dieses Resultat ist eine wichtige Erkenntnis für Studierende der Informatik. Oder sollte es zumindest sein ☺.

Kapitel 4.

Endliche Automaten und Reguläre Sprachen

Während mit Grammatiken Sprachen erzeugt werden können, bieten Automaten die Möglichkeit, für ein Eingabewort zu prüfen, ob das Wort zu einer bestimmten Sprache gehört. In der Informatik werden ganz unterschiedliche Automatenmodelle verwendet. Unterscheidungsmerkmale von Automaten sind dabei die Möglichkeiten zur Ein- und Ausgabe sowie die Größe des zur Verfügung stehenden Speichers.

In diesem Kapitel wird zunächst das einfachste Automatenkonzept vorgestellt, der endliche Automat. In den folgenden Kapiteln werden dann weitere Automatenkonzepte vorgestellt. Den endlichen Automat werden wir in unterschiedlichen Varianten betrachten. Angefangen bei dem deterministischen endlichen Automaten wird auch das wichtige Konzept des indeterminierten endlichen Automaten betrachtet. Wir werden sehen, dass der endliche Automat quasi das Pendant zur rechtslinearen Grammatik darstellt. In Satz 4.3.1 werden wir zeigen, dass zu jeder rechtslinearen Grammatik G ein endlicher Automat A konstruiert werden kann, so dass $L(G) = L(A)$, d.h. die durch die Grammatik G erzeugte Sprache ist identisch zur Sprache, die vom Automaten A erkannt werden kann. Diesen Sachverhalt: Grammatik zu Automat und Automat zu Grammatik werden wir in den folgenden Kapiteln auch für die anderen Sprachen der Chomsky-Hierarchie darstellen.

4.1. Einführendes Beispiel: Ein Getränkeautomat

Als einführendes Beispiel wollen wir einen einfachen Getränkeautomaten betrachten. Der Automat bietet eine gewisse Anzahl an Getränken, die jeweils einen individuellen Preis kosten. Wir starten mit einem ganz einfachen Automaten, der nur drei Getränke anbietet (Wasser, Apfelsaft, Orangensaft), die auch alle den identischen Preis von $1,50$ € kosten. Dabei soll der Automat unterschiedliche Geldmünzen entgegennehmen, jedoch kein Wechselgeld zurückgeben.

Ein derartiges Szenario lässt sich sehr schön mittels eines endlichen Automaten als System modellieren. Der Automat muss eine Folge von Eingaben des Anwenders entgegennehmen und diese dann verarbeiten. Nach Eingabe einer Folge von Geldstücken, die in Summe mindestens $1,50$€ betragen, muss der Automat die Getränkeauswahl anbieten, die Eingabe entgegen nehmen und das passende Getränk ausgeben. Die Verarbeitung möglicher Eingaben können wir mittels eines Zustandsdiagrammes (siehe Abbildung 4.1) darstellen.

Die Zustände des Automaten sind als Kreise dargestellt. Mit einem Übergang von einem Zustand zu einem anderen Zustand wird eine Nutzeraktivität beschrieben. Die Aktion zum Übergang von einem Zustand in den nächsten Zustand ist durch eine beschriftete Kante zwischen den beiden Zuständen dargestellt. Zu Anfang ist der Automat im Zustand start. Bei Eingabe

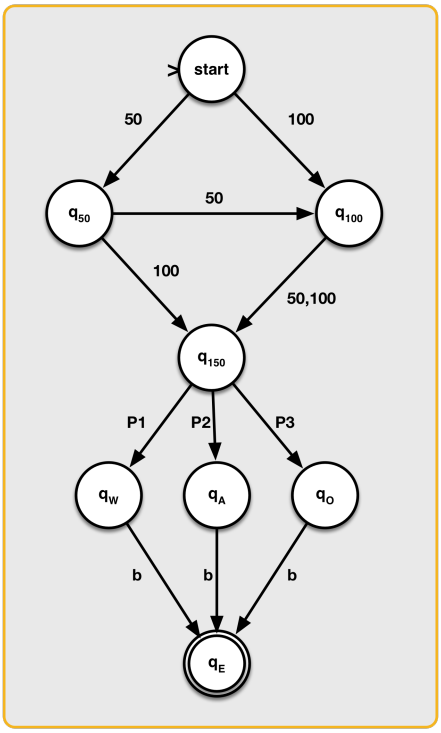


Abbildung 4.1.: Zustandsdiagramm des Getränkeautomaten

Zustand	Eingabe					
	50	100	P_1	P_2	P_3	b
$start$	q_{50}	q_{100}				
q_{50}	q_{100}	q_{150}				
q_{100}	q_{150}	q_{150}				
q_{150}			q_W	q_A	q_O	
q_W						q_E
q_A						q_E
q_O						q_E

Tabelle 4.1.: Übergangsfunktion des Getränkeautomaten

von $50ct$ geht der Automat in den Zustand q_{50}. Mit Hilfe dieses Zustands merkt sich der Automat, dass $50ct$ eingegeben wurden. Bei Eingabe weiterer oder anderer Münzen kann der Automat in die Zustände q_{100} oder q_{150} übergehen.

Die Kantenbeschriftung steht für eine Eingabe. $50ct$ an der Kante von Zustand q_{50} zu Zustand q_{100} bedeuten beispielsweise, dass insgesamt $100ct$ eingegeben wurden. Ist der Zustand q_{150} erreicht, so wurden insgesamt ausreichend viele Münzen eingeworfen und der Benutzer kann nun das gewünschte Getränk auswählen. Dazu führt der Benutzer eine entsprechende Auswahl P_1, P_2, P_3 aus und bestätigt dann die getroffene Auswahl.

Die in dem Zustandsdiagramm dargestellten Zustandsübergänge stellen das Programm oder die Ablaufvorschrift des Automaten dar. Dieses Programm, auch Übergangsfunktion genannt, kann auch einfach in einer Tabelle dargestellt werden. In dieser Tabelle gibt man an, mit welcher Eingabe von einem Zustand in einen anderen Zustand gewechselt wird. Die Tabelle 4.1 zeigt die Übergangsfunktion für den Getränkeautomaten.

Ganz allgemein können Automaten zur Erkennung von Sprachen eingesetzt werden. Die Sprache im obigen Beispiel des Getränkeautomaten ist die Eingabe des Benutzers, also die Eingabe der Geldmünzen oder die Durchführung einer Auswahl. Der Automat reagiert durch Zustandswechsel auf eine Folge von Eingaben und geht mit jeder Eingabe in einen neuen Zustand über. Bei den Zuständen wird unterschieden zwischen den finalen Zuständen (auch Endzustände) und den übrigen Zuständen. Ein Endzustand deutet an, dass die Eingabe korrekt oder gültig war. Im Beispiel des Getränkeautomaten ist der Zustand q_E ein Endzustand. Ein Eingabefolge zur Erreichung dieses Zustandes ist $(50, 100, P_1, b)$. Der Benutzer muss also zunächst hinreichend viele Münzen $(50, 100)$ einwerfen, dann eine Auswahl (P_1) tätigen und diese dann bestätigen (b).

Man kann sich einen Automaten also in abstrahierter Form wie in Abbildung 4.2 dargestellt vorstellen. Die Eingabe ist auf einem Eingabeband und wird einmalig ausgelesen. Das Lesen erfolgt durch einen Lesekopf auf dem Band und in Abhängigkeit des aktuellen Zustandes und der gelesenen Eingabe erfolgt ein Wechsel in den Nachfolgezustand. Akzeptiert ein Automat ein Eingabewort w, so gehört das Wort zu der vom Automaten akzeptierten Sprache. Die vom Getränkeautomaten akzeptierte Sprache sind die Eingaben

$$(50, 100, P_1, b),\ (50, 100, P_2, b),\ (50, 100, P_3, b),\ (50, 50, 50, P_1, b),\ (50, 50, 50, P_2, b),$$
$$(50, 50, 50, P_3, b),\ (100, 50, P_1, b),\ (100, 50, P_2, b),\ (100, 50, P_3, b),\ (100, 100, P_1, b),$$
$$(100, 100, P_2, b),\ (100, 100, P_3, b),\ (50, 50, 100, P_1, b),\ (50, 50, 100, P_2, b),\ (50, 50, 100, P_3, b).$$

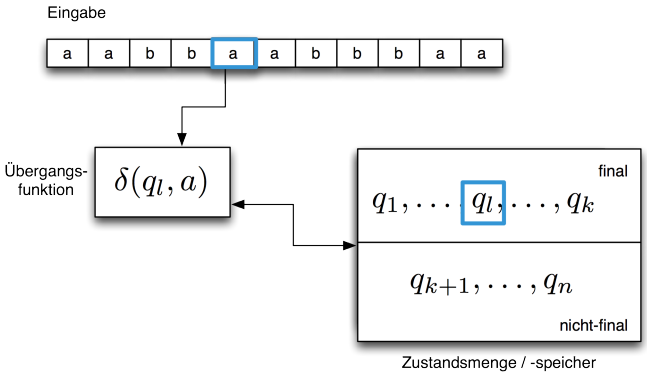


Abbildung 4.2.: Abstraktes Modell eines deterministischen endlichen Automaten

Erhält der Automat eine Eingabe, die nicht in einem finalen Zustand endet, so wird die Eingabe nicht akzeptiert, sie ist ungültig. Der Getränkeautomat ist das Beispiel eines endlichen Automaten. Endlich meint, dass der Automat endlich viele Zustände besitzt. Damit ist nicht die Endlichkeit der akzeptierten Eingaben gemeint. Wir werden schnell sehen, dass man auch mit endlichen Automaten unendlich viele Eingaben akzeptieren kann. Alle akzeptierten Worte werden als die vom Automaten akzeptierte Sprache bezeichnet.

Das Kapitel 4 behandelt die verschiedenen Varianten der endlichen Automaten. Die Menge aller Sprachen, die von endlichen Automaten akzeptiert werden, ist die Sprachklasse der rationalen oder regulären Sprachen. In diesem Kapitel werden auch die Zusammenhänge zwischen endlichen Automaten, Grammatiken und regulären Ausdrücken behandelt. Typische Fragestellungen sind in diesem Zusammenhang:

- Gegeben sei ein Wort w; wird das Wort von einem Automaten A akzeptiert?
- Gegeben sei eine Grammatik G; gesucht wird eine Automat A, der die Sprache $L(G)$ akzeptiert.
- Gegeben sei eine Sprache L; gesucht wird eine Automat A, der die Sprache L akzeptiert.

Die Lösung dieser Fragestellungen erfordert manchmal ein wenig Tüftelei, manchmal springt einem die Lösung aber förmlich auch ins Auge. Nicht abschrecken lassen sollte man sich dabei von der abstrakten und formalen Darstellung von Automaten. Durch die Einführung der graphischen Darstellung werden die Automaten sehr anschaulich.

4.2. Endliche Automaten

Endliche Automaten stellen das einfachste Automatenkonzept dar, welches wir im Rahmen der Veranstaltung behandeln werden. Sie verfügen über genau eine Speicherzelle, in der ein Zustand gespeichert werden kann. Wir werden in den folgenden Abschnitten unterschiedliche Konzepte der endlichen Automaten betrachten.

4.2.1. Endliche deterministische Automaten

Definition 4.2.1 (Endlicher Automat)
Ein endlicher Automat *(*DEA*) wird beschrieben durch ein Tupel*

$$A = (K, \Sigma, \delta, s_0, F)$$

Dabei ist

- K *eine endliche Menge von Zuständen*
- Σ *ein endliches Alphabet (aus dessen Zeichen die Eingabewörter bestehen können)*
- $\delta : K \times \Sigma \to K$ *die (totale) Übergangsfunktion*
- $s_0 \in K$ *der Startzustand*
- $F \subseteq K$ *die Menge der finalen Zustände*

Ein endlicher Automat bekommt eine Eingabe in Form eines Wortes $w \in \Sigma^*$ und verarbeitet es. Dies geschieht mit Hilfe der Übergangsfunktion δ. Die Interpretation dieser Übergangsfunktion ist wie folgt: Ist $\delta(q, a) = q'$, so liest der Automat im Zustand q das Zeichen a und geht damit in den Zustand q' über. Um mehrere Übergangsschritte des Automaten auf einmal betrachten zu können, erweitern wir δ zu δ^* in der folgenden Weise: $\delta^* : K \times \Sigma^* \to K$ ist induktiv über Σ^* definiert.

δ^*

$$\begin{array}{lcl} \delta^*(q, \epsilon) & := & q \\ \delta^*(q, wa) & := & \delta(\delta^*(q, w), a) \end{array}$$

Wenn klar ist, was gemeint ist, wird anstelle von δ^* einfach δ geschrieben. Abbildung 4.2 visualisiert die Elemente eines DEA.

Beispiel 4.2.2 *Es sei der folgende endlichen Automaten* $A = (\{q_0, q_1, q_2\}, \{a, b\}, \delta, q_0, \{q_0\})$ *gegeben, mit*

ToolBox$_{TI}$

$$\begin{array}{lcllll} \delta(q_0, a) & = & q_1 & \delta(q_1, a) = q_0 & \delta(q_2, a) = q_2 \\ \delta(q_0, b) & = & q_2 & \delta(q_1, b) = q_2 & \delta(q_2, b) = q_2 \end{array}$$

Um die Idee von δ^* zu verdeutlichen, berechnen wir für den Automaten den Wert der Übergangsfunktion mit der Eingabe aab, d.h. wir rechnen $\delta^*(q_0, aab)$.

$$\begin{array}{lcl} \delta^*(q_0, aab) & = & \delta(\delta^*(q_0, aa), b) \\ & = & \delta(\delta(\delta^*(q_0, a), a), b) \\ & = & \delta(\delta(\delta(\delta^*(q_0, \epsilon), a), a), b) \\ & = & \delta(\delta(\delta(q_0, a), a), b) \\ & = & \delta(\delta(q_1, a), b) \\ & = & \delta(q_0, b) \\ & = & q_2 \end{array}$$

Die Funktion δ^* formalisiert also gerade die Vorstellung davon, wie ein endlicher Automat arbeitet. Ein Wort - hier das Wort aab - wird von links nach rechts sukzessive Buchstabe für Buchstabe gelesen. Dabei wird entsprechend der δ-Regeln der Zustand mit jedem gelesenen Symbol verändert. $\delta^*(q, w) = p$ bedeutet also: wenn der Automat im Zustand q das Wort w liest, so ist er danach im Zustand p.

Definition 4.2.3 (Automat als Zustandsdiagramm)
Endliche Automaten können sehr anschaulich in Form eines Zustandsdiagrammes auch als Graph (siehe 2.4.1), bestehend aus Knoten und Kanten dargestellt werden.

- *Die Menge der Zustände des Automaten bildet die Knotenmenge des Graphen.*
- *Mit den Kanten repräsentiert man die Übergangsfunktion δ: Ist $\delta(q, a) = q'$, so gibt es im Graphen eine mit a beschriftete Kante vom Knoten q zum Knoten q'.*
- *Der Startknoten wird mit einem kleinen Pfeil gekennzeichnet, finale Zustände werden mit einem Doppelkreis markiert.*

Ist A ein endlicher Automat, so bezeichnen wir im folgenden mit $G(A)$ den auf diese Weise konstruierten Graphen zu A.

Beispiel 4.2.4 *Der Automat A aus Beispiel 4.2.2 ist als Graph $G(A)$ in Abbildung 4.3 zu sehen.*

ToolBox$_{TI}$

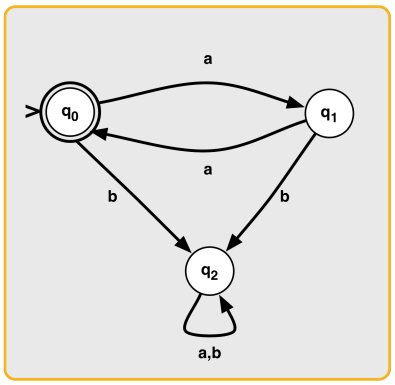


Abbildung 4.3.: Endlicher Automat A aus Beispiel 4.2.2 als Graph $G(A)$

Fehlerzustand

Im Graphen in Abbildung 4.3 ist auch ein häufig als *Fehlerzustand* bezeichneter Zustand zu erkennen: Der Zustand q_2 ist ein nicht-finaler Zustand und sobald der Automat einmal diesen Zustand erreicht hat, so kann er ihn nicht wieder verlassen, da keine Kante von q_2 zu einem anderen Zustand führt.

Ist ein Wort vom Automaten vollständig verarbeitet, so kann der Automat entweder in einem finalen Zustand sein oder eben in einem nicht finalen Zustand. Gelangt er für ein Eingabewort w in einen finalen Zustand, gilt also $\delta^*(s_0, w) \in F$, so hat der Automat das Wort erfolgreich verarbeitet oder *akzeptiert*. Ist $\delta^*(s_0, w) \notin F$, so ist das Wort nicht akzeptiert. Durch die folgende Definition wird die durch einen Automaten akzeptierte Sprache nun formalisiert.

Definition 4.2.5 (akzeptierte Sprache)

- *Die von einem deterministischen endlichen Automaten A* akzeptierte Sprache *$L(A)$ ist*

$$L(A) := \{w \in \Sigma^* \mid \delta^*(s_0, w) \in F\}$$

- *Die Menge der von endlichen Automaten akzeptierten Sprachen ist*

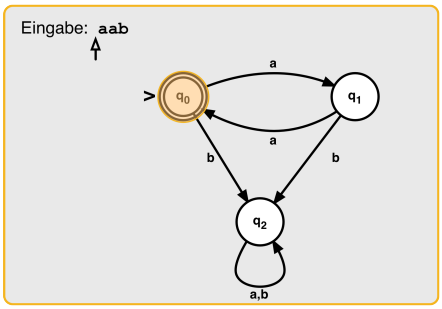
$$\mathcal{RAT} := \{L \mid \exists \text{ DEA } A \text{ mit } L = L(A)\}$$

und heißt die Menge der rationalen Sprachen.

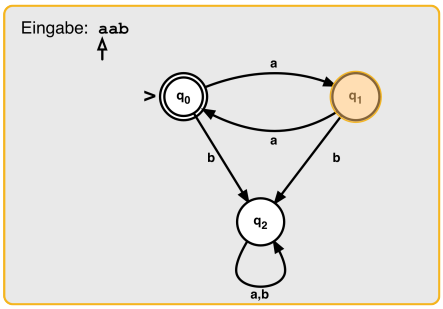
Ein Wort w wird von einem Automaten A nicht akzeptiert, wenn der Automat A angewendet auf w in einem nicht-finalen Zustand hält. Das bedeutet also:

$$\overline{L(A)} := \{w \in \Sigma^* \mid \delta^*(s_0, w) \notin F\}$$

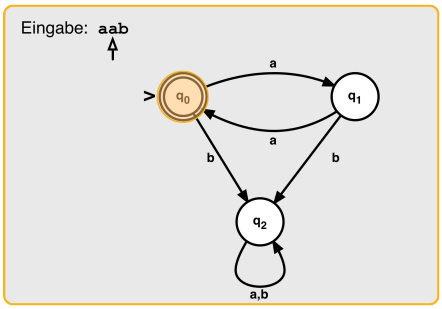
In der Abbildung 4.4 ist die Verarbeitung eines Wortes durch einen DEA dargestellt. In jedem Schritt wird ein Zeichen gelesen. Der aktuelle Zustand ist durch eine farbige Markierung angegeben. Das Eingabewort wird akzeptiert, wenn sich der Automat nach Lesen des letzten Zeichens in einem finalen Zustand befindet. Andernfalls wird es verworfen (oder nicht akzeptiert).



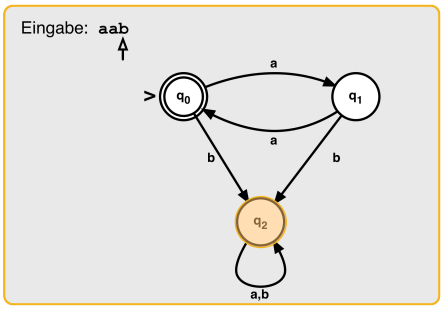
(a) Vor Verarbeitung



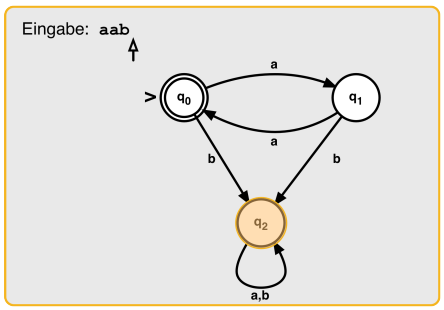
(b) Verarbeitung a



(c) Verarbeitung a



(d) Verarbeitung b



(e) Wort nicht akzeptiert

Abbildung 4.4.: Verarbeitung durch DEA

Die graphische Darstellung $G(A)$ eines Automaten A ist ein sehr nützliches und bequemes Hilfsmittel, um mit einem Automaten zu arbeiten. Es ist sehr viel intuitiver als die teilweise

unhandliche Definition der Übergangsfunktion. Der folgende Satz 4.2.6 stellt die Verbindung zwischen der Übergangsfunktion δ und einem Weg in dem zugehörigen Graphen her.

Satz 4.2.6 (Graph zum DEA)
Es sei $A = (K, \Sigma, \delta, s_0, F)$ ein deterministischer endlicher Automat. Es sei $G(A)$ der zugehörige Graph von A. Dann ist für alle $q_i, q_j \in K$ und $w \in \Sigma^+$ $\delta^*(q_i, w) = q_j$ genau dann, wenn es einen Weg mit Beschriftung w in $G(A)$ gibt, der vom Knoten q_i zum Knoten q_j führt.

Beweis

"$\Longrightarrow$": Es gelte also: für alle $q_i, q_j \in K$ und $w \in \Sigma^+$ $\delta^*(q_i, w) = q_j$. Zum Existenznachweis eines Weges führen wir einen Induktionsbeweis über die Wortlänge von w.

- Für Wortlänge 1 ergibt sich die Existenz direkt aus der Konstruktion des Graphen in 4.2.3.
- Nehmen wir also an, die Behauptung gelte für alle Wörter v mit $|v| \leq n$. Es sei nun $w = va$ ein beliebiges Wort mit Länge $n + 1$. Angenommen, nach der Verarbeitung des Teilwortes v erreicht der Automat den Zustand q_k, also

  $$\delta^*(q_i, v) = q_k.$$

  Da $|v| = n$ gibt es nach Induktionsannahme einen Weg in $G(A)$ mit Beschriftung v vom Knoten q_i zum Knoten q_k. Wenn nun nach Voraussetzung $\delta^*(q_i, w) = q_j$, dann muss für die Übergangsfunktion gelten: $\delta(q_k, a) = q_j$.

  Nach Konstruktion des Graphen 4.2.3 gibt es also einen Weg von q_i nach q_k mit Beschriftung v und es gibt eine Kante von q_k nach q_j mit Beschriftung a. Also gibt es insgesamt einen Weg von q_i nach q_j mit Beschriftung $w = va$. Damit ist dieser Teil der Behauptung nachgewiesen.

"$\Longleftarrow$": Für die andere Richtung des Beweises gelte nun, dass es einen Weg mit Beschriftung w von q_i nach q_j gebe. Es ist nun zu zeigen, dass dann auch $\delta^*(q_i, w) = q_j$ gilt. Auch hierfür führen wir einen Induktionsbeweis, nun über die Länge des Weges.

- Für Wege w mit $|w| = 1$ besteht der Weg gerade aus einer Kante von q_i nach q_j und nach Konstruktion des Graphen muss es dann eine Kante $\delta(q_i, w) = q_j$ geben.
- Die Behauptung gelte nun für alle Wege v mit $|v| \leq n$. Es sei nun $w = va$ ein Weg mit $|w| = n + 1$. Angenommen, nach Durchlaufen des Weges v wird im Graphen der Knoten q_k erreicht. Dann gilt nach Induktionsannahme

  $$\delta^*(q_i, v) = q_k$$

  und nach Konstruktion des Graphen muss es für das letzte Symbol eine Kante von q_k nach q_j mit Beschriftung a geben. Das bedeutet aber insgesamt:

  $$\delta^*(q_i, w) = \delta^*(q_i, va) = \delta(\delta^*(q_i, v), a) = \delta(q_k, a) = q_j.$$

Damit ist die Behauptung bewiesen.

■

Als Konsequenz erhalten wir somit das Lemma 4.2.7.

Lemma 4.2.7 *Für ein Wort $w \in \Sigma^*$ gilt $w \in L(A)$ genau dann, wenn ein Weg im Graph $G(A)$ von A mit der Beschriftung w existiert, der vom Startknoten zu einem finalen Knoten führt.*

Graphen sind also ein sehr anschauliches Hilfsmittel, um mit Automaten zu arbeiten. Wie einfach es sein kann, einen Automaten für eine spezielle Anforderung zu definieren, zeigen die folgenden Beispiele.

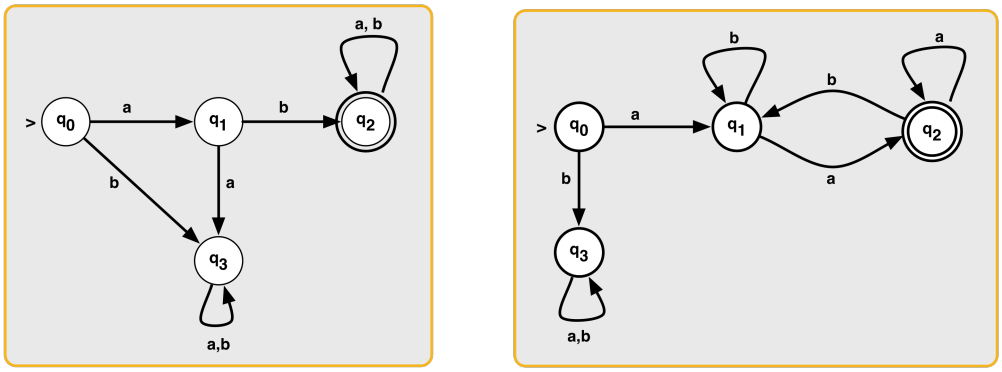
Beispiel 4.2.8 *Gesucht ist ein DEA zur Erkennung (=Akzeptanz) aller Worte auf $\Sigma = \{a, b\}$, die mit dem Präfix ab beginnen. Die relevante Information in einem Wort steckt in den ersten beiden Zeichen. Sind es die Buchstaben ab, so kann ein Wort akzeptiert werden. Sind sie es nicht, darf es nicht akzeptiert werden. Sind die ersten beiden Zeichen erfolgreich verarbeitet, muss der Rest des Wortes nur noch gelesen werden ohne weitere Auswertung und der Automat muss dann in einem finalen Zustand enden. Um einen möglicherweise auftretenden Fehler korrekt zu behandeln, wird der Automat mit einem* Fehlerzustand *ausgestattet. Ein solcher Fehlerzustand wird eingenommen, wenn sich die bislang vom Automaten verarbeitete Zeichenkette als nicht zu akzeptieren erweist. Dies ist in diesem Beispiel der Fall, wenn das erste Zeichen kein a ist oder wenn das zweite Zeichen kein b ist. Als Automat für die Sprache ergibt sich der in Abbildung 4.5(a) dargestellte Automat. Dabei ist der Zustand q_3 der Fehlerzustand.*

Fehlerzustand

ToolBox$_{TI}$

Beispiel 4.2.9 *Um die Frage zu beantworten, ob die Sprache $L = \{awa \mid w \in \{a, b\}^*\}$ eine rationale Sprache ist, muss ein endlicher Automat A mit $L = L(A)$ konstruiert werden. Ein solcher Automat muss prüfen, ob ein Wort mit einem a beginnt und mit einem a endet. Was zwischen dem Anfang und dem Ende liegt, spielt keine Rolle. Das Problem für die Konstruktion des Automaten liegt darin, dass ein Automat nicht vorausschauen kann, wann ein Wort beendet ist. Der Automat kann immer nur ein Zeichen lesen und dann entsprechend der Übergangsfunktion handeln. Er kann nicht erkennen, wieviele Zeichen nach dem aktuell gelesenen Zeichen noch kommen. Die Idee zur Lösung ist aber nicht so schwer. Nachdem das erste a gelesen ist, wechselt er beim Lesen des nächsten a in einen finalen Zustand. Folgen weitere a, so bleibt er in diesem. Wird aber ein b gelesen, so verlässt er den finalen Zustand und wechselt in einen nicht-finalen Zustand. Dies wird mit jedem gelesenen a so fortgesetzt. Die vollständige Lösung dieses Automaten ist in Abbildung 4.5(b) dargestellt.*

ToolBox$_{TI}$



(a) $L(A) = \{abv \mid v \in \Sigma^*\}$ (b) $L(a) = \{awa \mid w \in \{a, b\}^*\}$

Abbildung 4.5.: DEA zu Beispielen 4.2.8 und 4.2.9

Beispiel 4.2.10

(1) Der Automat aus Beispiel 4.2.2 akzeptiert die Sprache $L = \{a^{2n} \mid n \in \mathbb{N}_0\}$.

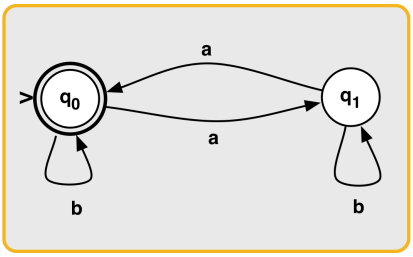
(2) Der Automat A' mit der akzeptierten Sprache

$$L(A') = \{w \in \{a, b\}^* \mid \exists n \in \mathbb{N}_0 : \#_a(w) = 2n\}$$

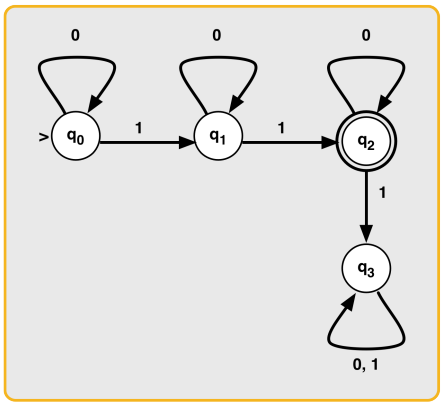
ist in Abbildung 4.6(a) dargestellt.

ToolBox$_{TI}$

ToolBox$_{TI}$ *(3) Die Sprache $L = \{w \in \{0,1\}^* \mid w$ enthält genau zwei Einsen$\}$ wird akzeptiert von dem Automaten in Bild 4.6(b).*



(a) DEA für die Sprache mit gerader Anzahl von a



(b) DEA für die Sprache mit den Worten, die genau zwei Einsen enthalten

Abbildung 4.6.: Beispiele endlicher Automaten

4.2.2. Indeterminierte endliche Automaten

Die Automaten aus Abschnitt 4.2 waren determinierte endliche Automaten. Das bedeutet, dass es in jedem Zustand q bei Lesen einer Eingabe a genau einen möglichen Folgezustand q' gibt mit $\delta(q,a) = q'$. Ein indeterminierter (auch nichtdeterministischer oder nichtdeterminierter) endlicher Automat kann hingegen mehrere Folgezustände haben oder auch gar keinen.

Definition 4.2.11 (indeterminierter endlicher Automat)

NDEA *Ein* indeterminierter endlicher Automat *(*NDEA*) A ist ein Tupel $A = (K, \Sigma, \Delta, I, F)$. Dabei ist:*

- K *eine endliche Menge von Zuständen*
- Σ *ein endliches Alphabet (aus dessen Zeichen die Eingabewörter bestehen können)*
- $\Delta \subseteq (K \times \Sigma) \times K$ *eine Übergangsrelation*
- $I \subseteq K$ *eine Menge von Startzuständen*
- $F \subseteq K$ *die Menge der finalen Zustände*

Im Vergleich zum deterministischen Automaten gibt es zwei Unterschiede: zum einen ist aus der Übergangsfunktion eine Übergangsrelation geworden. Als zweite Änderung gibt es beim indeterminierten Automaten nicht nur einen Startzustand sondern eine Menge I von Startzuständen.

Vereinbarung 4.2.12 *Wie auch bei den deterministischen Automaten in 4.2.3 definieren wir den zu einem NDEA A gehörigen Graph $G(A)$. Die Konstruktion erfolgt dabei vollkommen analog. Der Unterschied im Graphen zu einem DEA und einem NDEA besteht darin, dass ein Knoten in einem NDEA mehrere ausgehenden Kanten mit identischer Beschriftung besitzen kann und dass nicht für jedes Symbol des Alphabets Σ eine ausgehende Kante existieren muss.*

Da ein indeterminierter Automat eine Menge von Startzuständen besitzt, ist folglich in der graphischen Darstellung auch mehr als nur ein Startzustand gekennzeichnet. Als ein weiterer wichtiger Unterschied zum determinierten Automaten ist festzustellen, dass der Graph eines NDEA nicht notwendigerweise zusammenhängend ist. Der Zusammenhang des Graphen ergibt sich im Falle des DEA aus der Übergangsfunktion und im Falle des NDEA aus der Übergangsrelation. Da letztere nicht zwingend vollständig ist, ist die graphische Darstellung auch möglicherweise nicht zusammenhängend. Der zugehörige Graph zu einem NDEA kann somit aus mehreren Teilen bestehen. Ein Beispiel für einen nicht-zusammenhängenden Graphen ist beispielweise in Abbildung 4.14 auf Seite 66 zu sehen.

Beispiel 4.2.13 *Abbildung 4.7 zeigt einen indeterminierten Automaten, da es im Zustand q_0 zwei ausgehende Kanten mit Beschriftung a gibt. Ist der Automat also im Zustand q_0 und ist das aktuelle Eingabezeichen a, so kann der Automat in den Zustand q_1 oder in den Zustand q_4 wechseln.*

ToolBox$_{TI}$

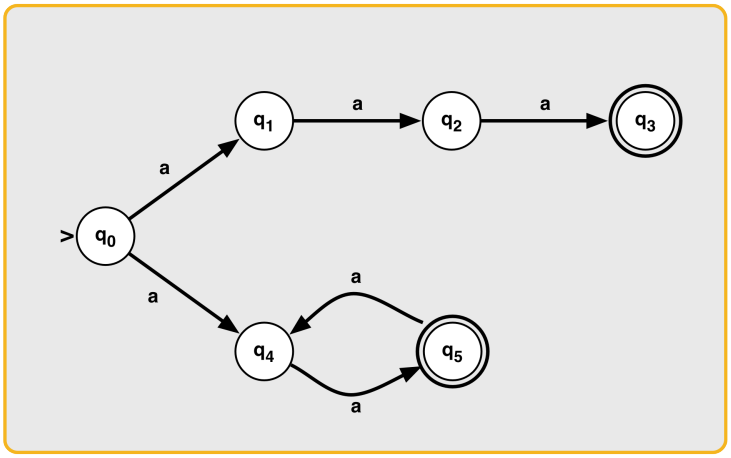


Abbildung 4.7.: Nichtdeterministischer endlicher Automat

Aus der Übergangsfunktion in der Definition des DEA (vgl. 4.2.1) ist nun eine Übergangsrelation geworden. Das bedeutet, ein NDEA hat in einem Zustand q und bei Lesen eines Zeichens a nicht nur eine Möglichkeit für den Übergang in den Folgezustand, sondern es kann durchaus mehre Möglichkeiten geben. Hat ein Automat mehrere Übergangsmöglichkeiten, so wählt der Automat zufällig eine Möglichkeit aus, er *errät* also den nächsten Zustand. Wie auch bei den determinierten Automaten erweitern wird zunächst die Übergangsrelation auf ganze Wörter: $\Delta^* \subseteq (K \times \Sigma^*) \times K$ ist induktiv definiert wie folgt:

$$\begin{array}{lcl} (q, \epsilon)\Delta^* q' & \iff & q' = q \\ (q, wa)\Delta^* q' & \iff & \exists q'' \in K : (q, w)\Delta^* q'' \text{ und } (q'', a)\Delta q' \end{array}$$

Anstatt $(q, w)\Delta^* q'$ schreibt man dabei auch $((q, w), q') \in \Delta^*$ oder $q' \in \Delta^*(q, w)$. Und zur Abkürzung schreiben wir Δ statt Δ^*, wenn es zu keiner Verwechslung kommen kann.

Mit der Vorstellung eines zufällig arbeitenden Automaten muss der Akzeptanzbegriff angepasst werden.

Definition 4.2.14 (Akzeptierte Sprache eines NDEA)
Die von einem indeterminierten Automaten A akzeptierte Sprache *ist*

$$L(A) := \{w \in \Sigma^* \mid \exists s_0 \in I, \exists q \in F : (s_0, w)\Delta^* q\} \tag{4.1}$$

Ein indeterminierter Automat A akzeptiert also ein Wort w, wenn es im zugehörigen Graphen $G(A)$ mindestens einen Weg mit der Beschriftung w gibt, der in einem finalen Zustand endet. Neben diesem einen "erfolgreichen" Weg darf es aber auch beliebig viele nicht erfolgreiche Wege geben, also Wege, die nicht in einem finalen Zustand enden. Man muss sicherstellen, dass der Automat auf keinen Fall nach dem Lesen eines Wortes w' in einem finalen Zustand ist, wenn w' nicht zur Sprache $L(A)$ gehört.

Man kann die indeterminierten Automaten auch so betrachten: Ist der Automat A in einem Zustand q und liest er die Eingabe a, so rät er, welcher der möglichen Folgezustände er annimmt. Gibt es nach vollständiger Verarbeitung eines Wortes keinen möglichen Folgezustand und ist der aktuelle Zustand nicht final, so wird das Eingabewort nicht akzeptiert.

Bemerkung 4.2.15 *Kann ein indeterminierter Automat ein Wort* $w = uv$ *nicht vollständig abarbeiten und bleibt beispielsweise nach der Verarbeitung des ersten Teilwortes* u *in einem Zustand* q *hängen, so ist* w *nicht akzeptiert, auch wenn* $q \in F$ *liegt. Per Definition der Übergangsrelation* Δ *ist in diesem Fall* $\Delta(s_0, w) = \Delta(s_0, uv) = \Delta(q, v) = \emptyset$*. Die Akzeptanz eine Wortes bedeutet aber, dass der erreichte Zustand in* F *liegen muss. Da aber per Definition der leeren Menge gilt* $\emptyset \cap F = \emptyset$*, ist das Wort nicht akzeptiert. Dies kann man in der grafischen Darstellung eines Automaten nicht erkennen, da hier das Hängenbleiben interpretiert wird als das Verweilen im erreichten Zustand! Und ist dieser final, wird das häufig fälschlicherweise als Akzeptanz interpretiert. Das ist aber falsch!*

Indeterminierte Automaten sind eine Verallgemeinerung der determinierten Automaten. Den Zusammenhang zwischen beiden Automatenkonzepten stellt die folgende Definition 4.2.16 her.

Definition 4.2.16 (determiniert, vollständig)
Ein indeterminierter endlicher Automat heisst determiniert*, wenn er beim Zustandswechsel nie eine Auswahlmöglichkeit hat. Er heisst* vollständig*, wenn er zu jedem Zustand und jedem Symbol des Eingabealphabets mindestens einen Folgezustand hat:*

$$A \textit{ heisst } \text{determiniert} \iff \forall a \in \Sigma\ \forall q \in K : |\Delta(q, a)| \leq 1 \tag{4.2}$$

$$A \textit{ heisst } \text{vollständig} \iff \forall a \in \Sigma\ \forall q \in K : |\Delta(q, a)| \geq 1 \tag{4.3}$$

Bemerkung 4.2.17 *Demnach ist ein endlicher Automat im Sinne der Definition 4.2.1 ein vollständiger und determinierter indeterminierter endlicher Automat.*

4.2.3. Erreichbare Zustände und Teilautomaten

Ein endlicher Automat kann Zustände enthalten, die von keinem Startzustand aus erreichbar sind. Solche Zustände tragen nichts zur akzeptierten Sprache des Automaten bei. Außerdem gibt es Zustände, von denen es aus nicht möglich ist, einen Endzustand zu erreichen. In indeterminierten Automaten können solche Zustände weggelassen werden.

Definition 4.2.18 (erreichbar, co-erreichbar, trim)
Sei $A = (K, \Sigma, \Delta, I, F)$*ein indeterminierter endlicher Automat. Ein Zustand* $q \in K$ *heißt:*

Teil III.

Anhang

Liste der YouTube-Videos

Die vollständige Liste ist bei YouTube als Playliste Theoretische Informatik abgelegt.

Kapitel 1 - **Einführung**

I	Lehrveranstaltungskonzept	15:32	https://youtu.be/X-jZHd_T1ZA
1.1	Peter Kruse über Revolutionäre Netze durch kollektive Bewegungen		https://youtu.be/e_94-CH6h-o

Kapitel 2 - **Grundlagen Mathematik**

2.1.14	Beispiel Relation	07:44	http://youtu.be/HldQVc2N79c
2.5.7	Abzählbarkeit von Σ^*	04:44	https://youtu.be/uz7T3FQcero
2.6.1	Prinzip des direkten Beweises	04:06	https://youtu.be/bS_e6DEpAUw
2.6.2	Prinzip des indirekten Beweises	06:20	https://youtu.be/i1qtaMmudC8
2.6.3	Prinzip der vollständigen Induktion	06:30	https://youtu.be/qbkCMbXvuqU
2.6.5	Prinzip der Diagonalisierung	05:35	https://youtu.be/43grR48BGtQ

Kapitel 3 - **Formale Sprachen**

2.5.1	Worte, Alphabete	04:52	https://youtu.be/LGQcEv_gDic
2.5.2	Operationen auf Worten	05:43	https://youtu.be/IV7DnJvsluM
2.5.3	Wortiteration, Reverse	05:03	https://youtu.be/E7ukn5HMiHs
2.5	Sprachen	06:54	https://youtu.be/swXcACNPDRw
3.1.1	Grammatiken	06:24	https://youtu.be/haf69V6GmdI
3.1.3	Ableitungen	05:12	https://youtu.be/OCmFG-uoteA
3.1.5	Sprache einer Grammatik	02:06	https://youtu.be/z4p4EkHnhEo
3.1.6	Sprache einer Grammatik, Beispiel	08:47	https://youtu.be/7fXIRWL-vLU
3.2.1	Rechtslineare Grammatik	04:42	https://youtu.be/_NmUQvRRf7U
3.2.1	Kontextfreie Grammatik	01:08	https://youtu.be/GiMvCcZ7Y0M
3.2.1	Kontextsensitive Grammatik	01:53	https://youtu.be/Axc5epAxcbA
3.2.1	Beschränkte Grammatik	07:00	https://youtu.be/DiyCzx-IIRs
3.2.10	Beispiel rechtslineare Grammatik	03:59	https://youtu.be/4vp30BCeyQU
	$L = \{ww^R \mid w \in \Sigma^*\}$	04:58	https://youtu.be/r5rI4G4AQVQ
	Clicker Peer Instruction	04:30	https://youtu.be/H61an5BP7Q4
3.1	Chomsky-Hierarchie	04:38	https://youtu.be/wjVhCQOcoO0

Kapitel 4 - **Endliche Automaten**

4.1	Getränkeautomat	12:43	https://youtu.be/ACwROThs8Vo
4.2.1	Definition DEA	06:24	https://youtu.be/Tn4FZLBRfRY
4.2.2	Beispiel DEA	05:56	https://youtu.be/udDUl76BDtQ
4.2.5	Sprache eines DEA	03:31	https://youtu.be/hyAqRI1DWAM
4.2.3	DEA als Graph	04:32	https://youtu.be/2EK6sFrFoQI
4.2.6	Graphische Darstellung und Sprache	14:35	https://youtu.be/4dAM1eRqIvg
4.2.8	Beispiel eines DEA	05:19	https://youtu.be/hE1szP3tThM
4.2.9	Beispiel eines DEA	05:38	https://youtu.be/d_6bBljtx40
4.2.11	Definition NDEA	04:25	https://youtu.be/H6nMpjq-1Pc
4.2.14	Sprache eines NDEA	07:33	https://youtu.be/sQ404MvLZVc
	Beispielanwendung Clicker	03:02	https://youtu.be/yA0L63FXANc
	Beispielanwendung Clicker	01:50	https://youtu.be/SkLImG4ZC3U
4.2.31	ϵ-NDEA	05:01	https://youtu.be/rn6HzxMK304
4.2.32	Sprache ϵ-NDEA	00:56	https://youtu.be/AeD-Mv_AoKs
4.2.33	Beispiel ϵ-NDEA	04:10	https://youtu.be/Nj7RKELYvss
4.2.34	Äquivalenz ϵ-NDEA zu NDEA	08:46	https://youtu.be/mdot4YOJiTc
4.2.35	Beispiel ϵ-NDEA zu NDEA	08:25	https://youtu.be/sFHGl-ciB7A
4.3.1	Äquivalenz $\mathcal{RAT} = \mathcal{L}_3$	12:54	https://youtu.be/YfcwJ1uJ9FY
4.3.2	Beispiel Umwandlung Grammatik zu ϵ-NDEA	02:45	https://youtu.be/WbhYuxjZslA

Kapitel 4 - **Abschlusseigenschaften von $\mathcal{L}_3$**

4.4.2	Abschlusseigenschaften $\mathcal{L}_3$	06:23	http://youtu.be/CiXvSh9Cthg
4.4.2	Abschluss $\mathcal{L}_3$ Komplement	03:20	http://youtu.be/3osGV1asbH8
4.4.2	Abschluss $\mathcal{L}_3$ Vereinigung	04:52	http://youtu.be/itReTe6isao
4.4.2	Abschluss $\mathcal{L}_3$ Konkatenation	02:07	http://youtu.be/LlNPXnfxfGM
4.4.2	Abschluss $\mathcal{L}_3$ Kleene-Stern	02:22	http://youtu.be/iG6yhKcDxzo
4.4.2	Abschluss $\mathcal{L}_3$ Durchschnitt	01:10	http://youtu.be/GNpKqKu2ATQ
4.4.2	Abschluss $\mathcal{L}_3$ Durchschnitt (Konstruktiv durch Produktautomat)	16:29	http://youtu.be/QNxfqosiOvE
4.4.6	Definition Homomorphismus	03:59	http://youtu.be/xy_A0DLZ9W4
4.4.8	Abschluss $\mathcal{L}_3$ Homomorphismus	04:44	http://youtu.be/k-EPG6Rs--0

Kapitel 4 - **Reguläre Ausdrücke**

4.5.1	Definition regulärer Ausdrücke	04:15	https://youtu.be/VyeKD03S7-g
4.5.2	Sprache eines regulären Ausdrucks	03:59	https://youtu.be/2B9nzrmpHXs
4.5.4	Beispiele regulärer Ausdrücke	05:40	https://youtu.be/NL_u0jnSSkk
4.5.8	Vorüberlegung $\mathcal{RAT} = \mathcal{RAT}$	11:47	https://youtu.be/EM5ExWWTQAs
4.5.9	Satz von Kleene $\mathcal{RAT} = \mathcal{RAT}$	28:22	https://youtu.be/5vHEBUVpvII
4.5.10	Vom DEA zum regulären Ausdruck	06:50	https://youtu.be/QS7OLZZlPI8

Kapitel 4 - **Pumping-Lemma**

4.6.3	Pumping-Lemma für $\mathcal{L}_3$	07:32	https://youtu.be/EVNDms9bTwY
4.6.2	Analyse des Pumping-Lemmas	02:05	https://youtu.be/f_c9KHovkV0
4.6.5	Beispiel Pumping-Lemma	09:20	https://youtu.be/V6aGSL4JdII
4.6.3	Checkliste Pumping-Lemma	01:37	https://youtu.be/s90pfByu9-A

Kapitel 4 - **Äquivalenz in Automaten**

4.7.1	Definition Relation $\sim_L$	03:55	https://youtu.be/nrQqYvW09cM
4.7.3	Beispiel zur Relation $\sim_L$	05:16	https://youtu.be/hBurZIGGUAY
4.7.4	Beispiel zur Relation $\sim_L$	12:19	https://youtu.be/GS7csnFsDxA
4.7.15, 4.7.19	Satz von Myhill-Nerode und Anwendungsbeispiel	05:02	https://youtu.be/ICM2Lsw513E
4.7.36	Minimierung DEA - Beispiel	11:37	https://youtu.be/b1J73lkdX3M
4.7.38	Minimierung endlicher Automaten	03:33	https://youtu.be/gwmFbQITIWk

Kapitel 4 - **Entscheidbare Probleme in $\mathcal{L}_3$**

4.8	Entscheidbare Probleme	05:19	https://youtu.be/6xKXuBP9Rqg

Kapitel 5 - **Kontextfreie Sprachen**

5.1	Kontextfreie Grammatik	https://youtu.be/6xKXuBP9Rqg
5.1.1	Beispiel DTD	https://youtu.be/cAop_EMd-jI
5.2	Ableitungsbaum	https://youtu.be/JjNVo7FrFdQ
5.2.1	Definition Ableitungsbaum	https://youtu.be/TvuNZwICsaY
5.2.4	Ordnung im Ableitungsbaum	https://youtu.be/kGtpHQ9w8pE
5.2.5	Ableitungsbaum vs. Ableitung	https://youtu.be/xKE8tXzGFjQ
5.2.5	Eindeutigkeit von Ableitung	https://youtu.be/s8HtSV1DeEw
5.2.7	Linksableitung	https://youtu.be/vJ9Casfe9bg
5.2.9	Mehrdeutigkeit	https://youtu.be/9GnJD6Y01sE
5.3.16	Umformung CFG	https://youtu.be/qQuwUTWwzZo
5.4.1	Chomsky-Normalform	https://youtu.be/qr9DezOdWVw
5.4.4	Greibach-Normalform	https://youtu.be/Z6d9t5V8lZM
5.6	PDA: Einführung	https://youtu.be/zrzCJDcHkbg
5.6.1	PDA: Definition	https://youtu.be/wjl4C4emuu4
5.6.2	PDA: Konfiguration	https://youtu.be/pAhq6FTVjjA
5.6.4	PDA: grafische Darstellung	https://youtu.be/xylkmszsOOA
5.6.6	PDA: akzeptierte Sprache	https://youtu.be/znDtVdei7UI
5.6.10	PDA: Finaler Zustand $\subseteq$ leerer Keller	https://youtu.be/gDyXaKKmWvg
5.6.11	PDA: leerer Keller $\subseteq$ Finaler Zustand	https://youtu.be/Paq2XoQ9Uto
5.6.12	PDA und $\mathcal{L}_2$	https://youtu.be/UIaKb7Wlme8
5.6.13	Beispiel zu PDA und $\mathcal{L}_2$	https://youtu.be/UIaKb7Wlme8#t=6m25s
5.5.1	Pumping-Lemma für $\mathcal{L}_2$	https://youtu.be/qAtXZqdDzNA
5.5.2	Beispiel Pumping-Lemma $\mathcal{L}_2$	https://youtu.be/SiWghgXzw7U
5.7	Abschlusseigenschaften $\mathcal{L}_2$	https://youtu.be/tY1VYbXfP0Y
5.8.1	Wortproblem in $\mathcal{L}_2$ und Idee CYK-Algorithmus	https://youtu.be/PbH9xVtyK84
5.8.2	Anwendungsbeispiel CYK	https://youtu.be/Q5TvCyu4RUo

Kapitel 6 - **Turing-Maschinen**

6.1.1	Definition	07:54	https://youtu.be/OZh5WSfqhTA
6.2	Arbeitsweise	02:01	https://youtu.be/fi65GNSRbk0
6.1.2	Konfiguration	04:11	https://youtu.be/584gNMOJyZo
6.1.3	Haltezustand	03:57	https://youtu.be/JZifzSYzzzc
6.1.5	Beispiel	05:02	https://youtu.be/Nmwd2FQA3vI
6.1.9	akzeptieren und entscheiden	04:49	https://youtu.be/6wS5NztOAK0
6.1.6	Akzeptieren von L_{ab}	19:11	https://youtu.be/C0xwokiWENc
6.1.7	Berechenbare Funktionen	02:28	https://youtu.be/bx7p0-uPJ3E
6.1.8	Rechnung auf $\mathbb{N}$	01:47	https://youtu.be/pDMNr2cCgZM
6.2.1	Grafische Darstellung TM	04:26	https://youtu.be/5D6Uj6_Z0yw
6.2.2	Beispiel Flussdiagramm	02:01	https://youtu.be/zDyl7C36lq0
6.2.3	Additionsmaschine	02:41	https://youtu.be/3DsQWVrhcV4
8.4.1	Zusammenhang Entscheidbarkeit und Akzeptanz	07:44	https://youtu.be/cxbF3oO8oas
8.4.3	rekursiv-aufzählbare Sprache	02:04	https://youtu.be/PJmDKT4W-oo
6.3.4	TM für $L_{ww} = \{w\#w \mid w \in \{0,1\}^*\}$	08:22	https://youtu.be/v6kT9Sy7DwU
6.3.5	TM für $L = \{a^i b^j c^k \mid i \times j = k\}$	11:03	https://youtu.be/kRZXF1cG7QU

Kapitel 6 - **Varianten von Turing-Maschinen**

6.4.2	zweiseitig unbeschränkte TM	01:44	https://youtu.be/OftzYR1Y5Jg
6.4.3	Satz: zwTM = TM	06:23	https://youtu.be/t9X6i03rSys
6.4.11	NTM: Beispiel aba als Teilwort		https://youtu.be/MbqiLoWGKNA
6.4.12	NTM: Beispiel Nicht-Primzahlen	05:50	https://youtu.be/HVEvrz9Nv24
6.4.8	NTM: Definition	01:09	https://youtu.be/PPzzptbvgGU
6.4.9	NTM: Konfiguration, Rechnung	01:20	https://youtu.be/cNZTrdEJk9Us
6.4.10	NTM: Halten, Akzeptanz	05:06	https://youtu.be/v9zx9N4JP1w
6.4.13	Satz: NTM = TM	20:23	https://youtu.be/acuUg6DdIQA
6.5	UTM: Einleitung	07:12	https://youtu.be/L3ynv2wvjyM
6.5.1	UTM: Gödelwort	06:26	https://youtu.be/x4FLBuOlEgM
6.5.2	UTM: Gödelisierung Beispiel	10:22	https://youtu.be/p9lFIZr-f3E
6.25	UTM: Arbeitsweise	02:33	https://youtu.be/D_4EhoqmCbo
6.5.3	Abzählbarkeit der TM	09:36	https://youtu.be/bHhDpEtrUrg

Kapitel 7 - **Sprachklassen $\mathcal{L}_0$, $\mathcal{L}_1$**

7.3.4	Gödelwort Grammatik	05:06	https://youtu.be/Ygh69V2NsPY
7.3.5	Abbildung Grammatik auf $\mathbb{N}$	02:09	https://youtu.be/JuBUT2erFKg
7.3.6	Beispiel Gödelwort zur Grammatik	02:35	https://youtu.be/XUVsT0TC6s4

Kapitel 8 - **Berechenbarkeit**

8.4.2	Entscheidbarkeit, Akzeptanz, Komplement	08:28	https://youtu.be/BIVRw3M8w2c
8.4.5	Akzeptierbare Sprache deren Komplement nicht akzeptierbar ist	15:14	https://youtu.be/jQGfQfQp2-Q
8.5.1	Einführung Halteproblem	01:55	https://youtu.be/1cb_pNUiQlw
8.5.2	Spezielles Halteproblem	06:17	https://youtu.be/DpkkQcl0ruk

Index

Literaturverzeichnis

[AB02] Alexander Asteroth and Christel Baier. *Theoretische Informatik*. Pearson Studium, 2002.

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: a modern approach*. Cambridge University Press, 2009. 259

[BS12] Jonathan Bergmann and Aaron Sams. *Flip your Classroom*. International Society for Technology in Education, 2012. 6

[EP02] Katrin Erk and Lutz Priese. *Theoretische Informatik*. Springer-Verlag Berlin Heidelberg New york, 2. edition, 2002. 159, 172, 173, 208, 220

[GL12] Jan Goyvaerts and Steven Levithan. *Regular Expressions Cookbook*. O'Reilly, 2nd edition, 2012. 91

[Goy13] Jan Goyvaerts. Regular expressions, Oktober 2013. 91

[Har02] David Harel. *Das Affenpuzzle und weitere bad news aus der Computerwelt*. Springer-Verlag Berlin Heidelberg New York, 2002.

[HMU11] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. Addison-Wesley (Pearson Studium), 3. edition, 2011. 6, 67, 139, 140, 205

[Hof09] Dirk W. Hoffmann. *Theoretische Informatik*. Hanser-Verlag, 2009. 6

[Hol07] Boris Hollas. *Grundkurs Theoretische Informatik*. Spektrum Akademischer Verlag, 2007. 272

[(Hr93] Hermann Engesser (Hrsg.). *Duden Informatik - Ein Sachlexikon für Studium und Praxis*. Duden Verlag, 2. edition, 1993. 9

[KPBS14] Lukas König, Friederike Pfeiffer-Bohnen, and Hartmut Schmeck. *100 Übungsaufgaben zu Grundlagen der Informatik*. Oldenbourg Verlag, 2014.

[Lag85] Jeffrey C. Lagarias. The 3x+1 problem and its generalizations. *Amer. Math. Monthly*, 92, 1985. 248

[Lin12] Peter Linz. *An Introduction to Formal Languages and Automata*. Jones Bartlett Learning, LLC, 5th edition, 2012.

[LP98] Harry Lewis and Christos Papdimitriou. *Elements of the theory of computation*. Prentice-Hall, 2nd edition, 1998. 6, 165

[Mor15a] Karsten Morisse. 2015.

[Mor15b] Karsten Morisse. Implementation of the inverted classroom model for theoretical computer science. In *Proceedings of E-Learn: World Conference on E-Learning in Corporate, Government, Healthcare, and Higher Education 2015*, pages 426–435, Kona, Hawaii, United States, October 2015. Association for the Advancement of Computing in Education (AACE). 6

[Pap95] Christos Papdimitriou. *Computational Complexity*. Addison-Wesley Publishing, 1995. 259

[PL04] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, 2004. 41, 42, 43

[Sip06] Michael Sipser. *Introduction to the Theory of Computation*. Thomson Course Technology, 2006. 205

[Soc08] Rolf Socher. *Theoretische Grundlagen der Informatik*. Hanser-Verlag, 3. auflage edition, 2008.

[Tur36] Alan Turing. On computable numbers with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265, 1936. 189

[Weg03] Ingo Wegener. *Komplexitätstheorie - Grenzen der Effizienz von Algorithmen*. Springer-Verlag Berlin Heidelberg New York, 2003. 259

[Weg05] Ingo Wegener. *Theoretische Informatik - eine algorithmenorientierte Einführung*. Leitfäden der Informatik. Teubner Verlag, 3. edition, 2005. 6, 67, 148

[WH09] Christian Wagenknecht and Michael Hielscher. *Formale Sprachen, abstrakte Automaten und Compiler*. Vieweg + Teubner, 2009.

[Win08] Adi Winteler. *Professionell lehren und lernen*. Wissenschaftliche Buchgesellschaft, 3. auflage edition, 2008. 8

[Woe10] Gerhard J. Woeginger. The p-versus-np page, November 2010. 272