Semesterarbeit

Evaluation von verschiedenen Ansätzen für Evolutionäre Algorithmen

Adrian Schmid

Zürich, 01.01.2014

Dank

Ich danke Bla und Bli

Abstract

This is rather abstract

Inhaltsverzeichnis

1 Einleitung

1.1 Ausgangslage

Evolutionäre Algorithmen sind Optimierungsverfahren die sich, inspiriert durch in der Natur ablaufende Prozesse, den grundlegenden evolutionären Mechanismen von Mutation, Selektion und Rekombination bedienen. In dieser Arbeit werden verschiedene Selektionsstrategien an einem einfachen Beispiel miteinander verglichen.

1.2 Ziel der Arbeit

Ziel der Arbeit ist es, an einem einfachen Beispiel, verschiedene Ansätze zum Entwurf von relativen Fitnessfunktionen gegeneinander abzuwägen. Dazu werden evolutionäre Algorithmen mit unterschiedlichen Strategien implementiert. Die aus den unterschiedlichen Ansätzen resultierenden Lösungen werden dann hinsichtlich ihrer Qualität und der für ihre Suche benötigten Ressourcen (Rechenzeit) verglichen.

1.3 Aufgabenstellung

In dieser Arbeit sollen evolutionäre Algorithmen implementiert welchen, welche zu einem gegebenen Regulären Ausdruck einen entsprechenden endlichen Automaten finden. Für dieses konkrete Beispiel sollen evolutionäre Algorithmen mit verschiedenen Strategien implementiert werden, umso das Konvergenzverhalten und die Rechenzeit der verschiedenen Implementationen (und somit der Strategien) untersuchen und vergleichen zu können.

- Einarbeitung in das Thema
- Repräsentation und Mutation von endlichen Automaten definieren
- Evolutionärer Algorithmus mit fester, globaler Problemmenge implementieren
- Evolutionärer Algorithmus mit mutierender, globaler Problemmenge implementieren
- Evolutionärer Algorithmus, bei welchem jeder Lösungskandidat seine eigene, mutierende Problemmenge mitführt implementieren
- Gegenüberstellung der Resultate

1.4 Erwartete Resultate

- Überblick über Evolutionäre Algorithmen und Endliche Automaten im technischen Bericht
- Implementierung einer Repräsentation von endlichen Automaten, Implementierung von Mutationen zu endlichen Automaten und Dokumentation des Implementierten im technischen Bericht
- Implementierung eines evolutionären Algorithmus mit fester, globaler Problemmenge mit einer entsprechenden Dokumentation im technischen Bericht
- Implementierung eines evolutionären Algorithmus mit mutierender, globaler Problemmenge mit einer entsprechenden Dokumentation im technischen Bericht
- Implementierung eines evolutionären Algorithmus, bei welchem jeder Lösungskandidat seine eigene, mutierende Problemmenge mitführt mit einer entsprechenden Dokumentation im technischen Bericht
- Dokumentation der Gegenüberstellung der Resultate

2 Grundlagen

2.1 Endliche Automaten und reguläre Sprachen

Reguläre Sprachen und *endliche Automaten* gehören zur Automatentheorie, einem Teilbereich der theoretischen Informatik. Zum besseren Verständnis dieser Arbeit habe ich folgend einige gängige und wichtige Konzepte der Automatentheorie frei nach dem Standartwerk von Hopcroft, Motwani und Ullman zusammengefasst. [2]

Alphabet Ein Alphabet ist eine endliche, nicht leere Menge von Symbolen. Üblicherweise wird ein Alphabet durch das Symbol Σ dargestellt.

Zeichenreihen Eine Zeichenreihe (auch Wort oder String) ist eine endliche Folge von Symbolen eines bestimmten Alphabetes.

Sprachen Eine Menge von Zeichenreihen aus Σ^*, wobei Σ ein bestimmtes Alphabet darstellt, wird als Sprache bezeichnet. Das heisst, eine Sprache ist eine Menge von Zeichenreihen die mit den Zeichen aus einem Alphabet Σ gebildet werden können. Bei der Bildung von Wörtern müssen nicht alle Zeichen des gegebenen Alphabets verwendet werden.

Bei der Klasse der *regulären Sprachen* handelt es sich um die Menger aller Sprachen, welche durch einen endlichen Automaten beschrieben werden können. Alle Sprachen dieser Klasse haben auch die Eigenschaft, dass Sie durch einen regulären Ausdruck dargestellt werden können. Das heisst, dass es für jede reguläre Sprache sowohl einen regulären Ausdruck als auch einen endlichen Automaten gibt, der alle Wörter der Sprache akzeptiert.

Ein deterministischer endlicher Automat besteht gemäss Hopcroft, Motwani und Ullman [2] aus:

1. einer endlichen menge von Zuständen die meist durch Q bezeichnet wird.
2. einer endlichen Menge von Eingabesymbolen (auch Alphabet Σ).
3. einer Übergangsfunktion δ der ein Zustand und ein Eingabesymbol übergeben werden und die einen Zustand zurückgibt.
4. einem Startzustand welcher Teil der Menge Q sein muss.
5. einer Menge F akzeptierender Zustände. Die Menge F ist eine Teilmenge von Q.

Oft werden endliche Automaten als Quintupel in der Form $A = (Q, \Sigma, \delta, q0, F)$ dargestellt.

Zum besseren Verständnis folgend ein Beispiel eines Automaten welcher alle Binärzahlen die durch drei Teilbar sind akzeptiert.

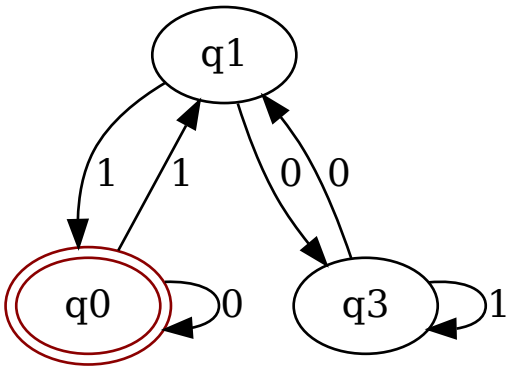

Abbildung 1: Beispiel: Endlicher Automat

Die Attribute des Quintupels für diesen Automaten sind:

1. $Q = \{q0, q1, q3\}$
2. $\Sigma = \{0, 1\}$
3. δ
4. $q0$
5. $F = \{q0\}$

Wobei man die Übergangsfunktion δ am einfachsten als Tabelle wie folgt darstellt:

	0	1
→q0*	q0	q1
q1	q3	q0
q3	q1	q3

Tabelle 1: Übergangstabelle

Konventionen Ich habe mich bei der Darstellung von Automaten im Rahmen dieser Arbeit auf folgende Konvention festgelegt:

- Zustände werden mit einem kleinen q gefolgt von einer Zahl bezeichnet (zB. $q1$)
- Zustände müssen nicht durchgängig nummeriert sein
- Zustände werden als Ellipsen dargestellt
- Zustandsübergänge sind beschriftete Pfeile
- der Startzustand erhält einen roten Rand
- alle akzeptierenden Zustände sind doppelt umrandet

2.2 Evolutionäre Algorithmen

'Unter evolutionären Algorithmen (EA) verstehen wir randomisierte Heuristiken, die Suchprobleme näherungsweise durch vereinfachende algorithmische Umsetzung von Prinzipien der natürlichen Evolution zu lösen versuchen. Somit geben evolutionäre Algorithmen in der Regel weder eine Garantie bzgl. der benätigten Rechenzeit noch der Güte der ausgegebenen Lösung. Ein Suchproblem besteht darin, zu einer Zielfunktion ein Element aus deren Definitionsbereich zu finden, dessen Funktionswert möglichst gut ist. Darunter verstehen wir im Folgenden, wenn nicht ausdrücklich anders vermerkt, einen möglichst grossen Funktionswert, weshalb der Zielfunktions wert eines Elements auch als seine Fitness bezeichnet wird.

Der Aufbau eines evolutionären Algorithmus lässt sich dann grob wie folgt beschreiben: in jedem Schritt verwaltet er eine Menge fester Grösse von Suchpunkten, die so genannte Population, wobei jeder einzelne Suchpunkt auch als Individuum bezeichnet wird. Aus den Punkten der Population neue Punkte zu erzeugen, ist Aufgabe von Mutation und Rekombination. Dabei steht hinter der Mutation die Idee, jeweils nur ein einzelnes Individuum zufällig zu verändern, ohne dass andere Individuen dabei berücksichtigt werden. Durch Rekombination wird hingegen aus mehreren, meist zwei Individuen zufällig ein neues gebildet, das von diesen möglichst gute Eigenschaften übernehmen soll. Durch Mutation und Rekombination werden also neue Individuen (Kinder genannt) aus bestehenden Individuen (Eltern genannt) erzeugt. Beide Operatoren hängen oftmals stark von Zufallsentscheidungen ab. Jedoch fliesst in der Regel weder in Mutation noch Rekombination der Zielfunktionswert der Individuen ein.

Die Zielfunktion beeinflusst nur die Selektion. Dieser Operator wählt Individuen der Population aus, sei es zur Auswahl der Eltern für eine Rekombination oder Mutation oder, um aus der Menge von Eltern und Kindern die nächste Population zu wählen, was den Übergang zur nächsten Generation darstellt. Dadurch, dass die Selektion Punkte

mit höherem Zielfunktionswert mit grösserer Wahrscheinlichkeit auswählt, soll erreicht werden, dass nach und nach immer bessere Punkte gefunden werden.' [1]

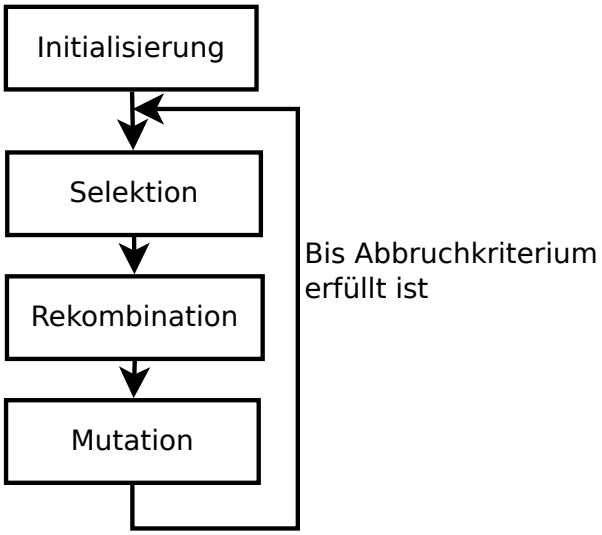

Abbildung 2: Evolutionärer Algorithmus

2.2.1 Anwendung auf unser Problem

Unser Problem der Findung von regulären Automaten welche die selbe Sprache akzeptieren wie ein gegebener regulärer Ausdruck läst sich wie mit einigen Anpassungen in diesem Standartmodell eines evolutionären Algorithmus abbilden.

Initialisierung In der Initialisierungsphase werden wir eine Population von Automaten zufällig erzeugen. Das heisst wir erzeugen eine Zufällige Anzahl von Zuständen, verbinden diese Zufällig und setzen einen zufälligen Start und zufällige Akzeptierende Zustände.

Selektion Es werden jeweils diejenigen Automaten selektiert welche eine höhere Fitness aufweisen. Es werden verschiedene Ansätze zur Selektion implementiert und es wird deren Konvergenzverhalten analysiert.

Fitness Die Fitness von Automaten bestimmen wir in dem wir ihn mit Wörtern füttern und die Ausgabe mit der soll Ausgabe vergleichen. Die Soll Ausgabe können wir anhand des gegebenen regulären Ausdrucks einfach bestimmen. Der Wert der Fitnessfunktion entspricht dann der Summe der richtig akzeptierten und richtig nicht akzeptierten Wörtern aus unserer *Problemmenge*.

Problemmenge Die Problemmenge ist eine Menge von Tupeln ($Wort$, $SollAusgabe$) welche verwendet wird um die Fitness von Automaten zu bestimmen. Im Rahmen dieser Arbeit werden unter anderem Verschiedene Ansätze zur Erzeugung und zum Verhalten solcher Problemmengen ausprobiert. Dabei wird das Konvergenzverhalten des evolutionären Algorithmus untersucht.

Rekombination Wenn man zwei endliche Automaten graphisch zusammenfügt, wird der resultierende Automat ein komplett anderes Verhalten aufweisen als die beiden Eltern. Deshalb verzichten wir auf die rekombination im herkömmlichen Sinne. Wir klonen jeweils die selektierten Automaten und *mutieren* die so entstandenen Kinder.

Mutation Bei der Mutation soll jeweils eine zufällige Veränderung an einem Automaten durchgeführt werden. Wir haben uns auf folgende Mutationen festgelegt:

1. einen Zustand hinzufügen
2. einen Zustand entfernen
3. eine Verbindung zwischen zwei Zuständen ändern
4. einen nicht akzeptierenden Zustand zu einem akzeptierenden Zustand umwandeln
5. einen akzeptierenden Zustand zu einem nicht akzeptierenden Zustand umwandeln

3 Umsetzung

3.1 Automaten

Es wurde versucht Automaten sowohl möglichst nah an der theoretischen Vorgabe als auch möglichst effizient zu implementieren. Herausgekommen ist dabei folgende Grundimplementation:

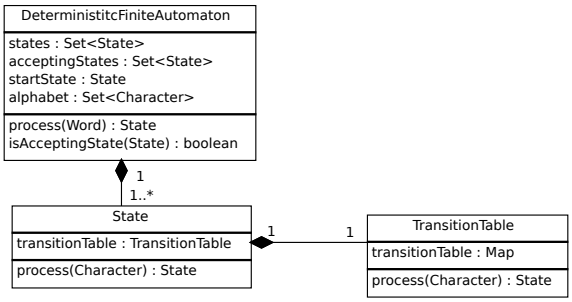

Abbildung 3: DFA Klassendiagramm vereinfacht

Das Quintupel (Q, Σ, δ, $q0$, F) ist darin wie folgt abgebildet:

Q	Die Menge der Zustände wurde als Set von States implementiert.
Σ	Das Alphabet ist ein Array von Zeichen (Character).
δ	Die Übergangsfunktion δ wird als Map (Zeichen -> Zustand) auf den einzelnen Zuständen abgebildet. Dies erlaubt uns eine einfache und effiziente Verarbeitung von Eingaben sowohl auf Zustands als auch auf Automaten Ebene.
$q0$	Eine Referenz zum Startzustand $q0$ ist in der DFA Klasse hinterlegt.
F	Die Menge der Akzeptierenden Zustände wird durch ein Set auf dem jeweiligen DFA Objekt representiert.

Tabelle 2: Implementation automaten

Ein solcher Automat kann nun mithilfe seiner $process(Word)$ Funktion ein Wort (Eine Liste von Zeichen) verarbeiten indem er sich die Referenz des ersten Zustandes $startState$ holt und dann Zeichen für Zeichen jeweils mit $process(Character)$ die Referenz des nächsten Zustandes herausliest. Zurückgegeben wird schlussendlich derjenige Zustand der am Ende der verarbeiteten Zeichenkette erreicht wurde. Mithilfe der $isAcceptingState(State)$ Methode könnte man nun feststellen ob das Wort akzeptiert wird oder nicht.

Zum besseren Verständnis folgend die Process Methode der DFA Klasse als Pseudocode:

```
def process(word):
  state = startState
  for char in word:
    state = state.process(char)
  return state
```

Listing 1: Process Methode der DFA Klasse

3.1.1 RandomDeterministicFiniteAutomaton

Der *RandomDeterministicFiniteAutomaton* ist ein erweiterter DFA welcher den normalen endlichen Automaten um einen zufälligen Konstruktor und eine *mutate* Methode zum zufälligen mutieren des Automaten erweitert.

Der Konstruktor erzeugt mithilfe eines gegebenen Alphabets und einem Faktor für die Komplexität des Problems - wie folgt beschrieben - zufällige Automaten.

1. Es wird eine Menge von Zuständen erzeugt (Anzahl Zustände: 1 - (2·$AnzahlZeichen$·$Komplexitaet$) wobei $AnzahlZeichen$ die Anzahl der Zeichen des Alphabets und die $Komplexitt$ ein Faktor ist welcher als Parameter an den Konstruktor übergeben wird).
2. Jedem Zustand werden für jedes Zeichen des Alphabets zufällig Verbindungen auf andere Zuständen zugeordnet.
3. Der erste Zustand (q0) wird zum Startzustand.
4. Um sicherzustellen dass jeder Automat erreichbare akzeptierende Zustände hat, werden alle nicht erreichbaren Zustände entfernt.
5. Eine zufällige Menge von Zuständen (1 - $\frac{AnzahlZustaende}{5}$) wird zu akzeptierenden Zuständen.

Die *mutate* Methode greift auf ein Mutationsregister zurück in welchem die Methoden zum Verändern von Automaten registriert sind. Dabei wählt es zufällig eine Methode aus und führt diese durch. Wenn die Mutation erfoglreich war sind wir fertig. Wenn nicht wird erneut eine Zufällige Methode ausgesucht. Dies wird solange wiederholt bis der Automat erfolgreich verändert wurde.

Aktion	**Beschreibung**	**Bedingungen**
Zustand hinzufügen	Es wird ein Zustand zum automaten hinzugefügt. Für jedes Zeichen im Alphabet wird ein Übergang auf einen zufälligen Zustand des Automaten gelegt. Danach wird berechnet wieviele Verbindungen in diesem Automaten durchschnittlich zu einem Zustand führen (avg). Mit diesem Resultat werden zwischen $\frac{avg}{2}$ und $avg \cdot 2$ zufällig ausgewählte Verbindungen von zufällig ausgewählten Zuständen zum neuen gelegt.	-
Zustand entfernen	Es wird zufällig ein Zustand ausgewählt und entfernt. Alle Übergänge die auf diesen Zustand zeigen werden zufällig auf andere Zustände umgeleitet.	Nicht der letzte Zustand; Nicht der einzige akz. Zustand
Akz. Zustand hinzufügen	Es wird ein zufälliger nicht akzeptierender Zustand ausgewählt. Dieser wird zum akzeptierenden Zustand gemacht.	Nicht akzeptierende Zustände vorhanden
Akz. Zustand entfernen	Es wird ein zufälliger akzeptierender Zustand ausgewählt. Dieser wird zu einem regulären nicht-akzeptierenden Zustand umgewandelt.	Nicht der einzige akz. Zustand
Übergang ändern	Es wird ein zufälliger Zustand und ein zufälliges Zeichen ausgewählt. Der abgehende Übergang für das gewählte Zeichen wird zufällig auf einen anderen Zustand gelegt.	-

Tabelle 3: Mutationen

3.1.2 Ausgabe

Die grafische Ausgabe der Automaten wurde mithilfe von Graphviz [**?**] umgesetzt. Um eine Grafik mit Graphviz zu erzeugen geht man normalerweise in zwei Schritten vor. Zuerst schreibt man das Beschreibungsfile welches festlegt wie die Grafik aussehen soll. Danach ruft man einen Kommandozeilenbefehl auf welchem man das Beschreibungsfile und das Ausgabeformat mitgibt. Dieser Befehl erzeugt dann die Grafik.

Laszlo Szathmary hat eine kleine aber feine Java Klasse implementiert, welche das Ausgabeformat, den Zielpfad und die Grafikbeschreibung als String entgegennimmt und daraus direkt die Grafik am angegebenen Pfad erstellt. Diese Klasse wurde in diesem Projekt als API verwendet um einfach Grafiken erstellen zu können.

In unserer Applikation wurde ein Interface *GraphvizRenderable* definiert, welches die Methode *generateDotString*() vorschreibt. Zudem gibt es eine einfache Klasse *GraphvizRenderer* welche eine statische Methode $renderGraph(GraphvizRenderable, FileName)$ besitzt. Diese Methode ruft *generateDotString*() auf dem mitgegebenen Objekt auf und generiert damit und mithilfe der Klasse von Laszlo Szathmary eine Vektor-Grafik (svg) am angegebenen Ort ($FileName$).

3.1.3 Überprüfung

Um ein Resultat unserer evolutionären Algorithmen auf seine korrektheit zu prüfen, wird eine Methode zum Vergleichen von Automaten und regulären Ausdrücken benötigt, welche prüft ob sowohl der Automat als auch der reguläre Ausdruck die selbe Sprache akzeptieren.

Die Komplexität und die Menge der Algorithmen die benötigt würde um dies zu erreichen sprengt den Rahmen dieser Arbeit. Um dieses Problem zu Überbrücken wird auf die Library von BRICS zurückgegriffen. Diese Library beherrscht neben anderen Dingen unter anderem:

- das Erzeugen von Automaten aus Regulären Ausdrücken
- einen Vergleich von Automaten welcher überprüft ob beide die selbe Sprache akzeptieren

Um dies verwenden zu können, wurde ein $Transformer < Source, Target >$ Typ definiert, welcher ein Objekt vom Typ *Source* in ein Objekt des Typen $Target$ umwandelt. Die konkrete Implementation um unsere Automaten in Brics Automaten zu verwandeln heisst $TransformDFAToBricsAutomaton$. Sie erstellt einen leeren BRICS Automaten und fügt dann in einem ersten Schritt erstmal alle Zustände aus dem *Source* Automaten in diesen ein. In einem zweiten Schritt werden dann alle Zustandsübergänge übertragen.

Um nun einen Automatne zu prüfen wird einfach ein Referenzautomat aus dem entsprechenden regulären Ausdruck generiert. Dieser (BRICS) Automat wird dann mit dem Umgewandelten Lösungskandidaten verglichen (Vergleich auf akzeptierende Sprache) um herauszufinden ob wir eine gültige Lösung gefunden haben.

3.1.4 Entfernen nicht erreichbarer Zustände

Vor der Ausgabe und nach dem Erzeugen werden nicht erreichbare Zustände entfernt um a) die Ausgabe einfacher zu halten und b) zu verhindern, dass wir Automaten erzeugen mit nicht erreichbaren akzeptierenden Zuständen. Um nicht erreichbare Zustände zu entfernen starten wir am Startzustand und fü

```
def optimise(Automaton obj):
  #1. Initialisieren
  queue = Set([])
  nextList = Set([])
  processed = Set([])

  current = obj.startState
  queue.add(current)

  #2. Erreichbare Zust nde finden
  while len(queue) > 0:
    nextList.clear()
    for c in obj.alphabet:
      next = current.process(c) #Zustand finden, welcher vom aktuellen aus
    mit dem Buchstaben c erreicht wird
      nextList.add(next)

    processed.add(current)
    queue.remove(current)

    if len(queue) > 0:
      current = queue.get(0)

  newStates = processed
  newAcceptingStates = Set([])

  #3. Set der akzeptierenden Zust nde auf die Erreichbaren limitieren
  for s in obj.acceptingStates:
    if(newStates.contains(s)):
      newAcceptingStates.add(s)

  obj.setStates(newStates)
  obj.setAccepctingStates(newAcceptingStates)
```

Listing 2: Process Methode der DFA Klasse

3.2 Evolutionäre Algorithmen

3.2.1 Feste globale Problemmenge

3.2.2 Mutierende globale Problemmenge

3.3 Lokale mutierende Problemengen

4 Supi5

Protokoll Kick-Off Meeting

Steht die Auftraggeberin bzw. der Auftraggeber hinter dieser Semesterarbeit?
Ja

Sind die fachliche Kompetenz und die Verfügbarkeit der Betreuungsperson sichergestellt
Ja

Sind die Urheberrechte und Publikationsrechte geklärt?
Ja

Bekommt die Studentin oder der Student die notwendige logistische und beratende Unterstützung durch die Auftraggeberin bzw. den Auftraggeber?
Es exitistert kein externer Auftraggeber

Entsprechen Thema und Aufgabenstellungen den Anforderungen an eine Semesterarbeit?
Ja

Ist die Arbeit thematisch klar abgegrenzt und terminlich entkoppelt von den Prozessen (des Unternehmens) der Auftraggeberin bzw. des Auftraggebers?
Ja

Ist eine Grobplanung vorhanden? Sind die nächsten Schritte klar formuliert (von der Studentin oder dem Studenten)?
Nächste Schritte:

- Implementation von endlichen Automaten in Java
- Implementation von Automaten-Mutationen in Java
- Implementation grafischen Darstellung von endlichen Automaten
- Implementation der evolutionären Algorithmen
- Testen der Konvergenzverhaltne der verschiedenen implementationen

Dazu parallel:

- Dokumentieren des erarbeiteten Wissens / erstellen des technischen Berichts.

Ist die Arbeit technisch und terminlich von der Studentin oder dem Studenten umsetzbar?
Ja

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[1] Stefan Droste. *Zu Analyse und Entwurf evolutionarer Algorithmen.* Universität Dortmund, Fachbereich Informatik, 2000.

[2] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie.* Pearson Education, 2002.