

Phrasendreschmaschine und Text-Evolution: Unterrichtsideen für Zeichenketten mit PYTHON

Reinhard Oldenburg

Angaben zur Veröffentlichung / Publication details:

Oldenburg, Reinhard. 2008. "Phrasendreschmaschine und Text-Evolution: Unterrichtsideen für Zeichenketten mit PYTHON." Log in: Informatische Bildung und Computer in der Schule, no. 154/155: 91-98.

Nutzungsbedingungen / Terms of use:

licgercopyright

Dieses Dokument wird unter folgenden Bedingungen zur Verfügung gestellt: / This document is made available under the following conditions:

Deutsches Urheberrecht

Weitere Informationen finden Sie unter: / For more information see:

<https://www.uni-augsburg.de/de/organisation/bibliothek/publizieren-zitieren-archivieren/publizieren>



Phrasendreschmaschine und Text-Evolution

Unterrichtsideen für Zeichenketten mit PYTHON

von Reinhard Oldenburg

Die Datenstruktur der Zeichenketten sollte in allen Informatikkursen behandelt werden, in denen das Entwickeln und Erproben von Programmen Unterrichtsinhalt ist. In diesem Beitrag werden deshalb einige Anwendungen vorgestellt, die mit wenigen Grundlagen der Algorithmik auskommen und diese üben, andererseits aber auch Vergnügen bereiten und Relevanz besitzen. Im zweiten Heft des neunten Jahrgangs (1989) wurde in dieser Zeitschrift (im LOG OUT auf Seite 64) die Frage gestellt: „Wie würden heute solche Programme [wie das „Weihnachtsgedicht“, siehe Kasten „Stochastische Texte“] aussehen, und wie könnte Unterricht aussehen, der dieses Thema aufgreift?“ Der vorliegende Beitrag gibt darauf eine (leicht verspätete) Antwort.

Bezüglich prozessbezogener Kompetenzen sind vor allem die Bereiche *Modellieren und Implementieren* sowie *Kommunizieren und Kooperieren* am Kompetenzaufbau beteiligt. Die Schüler und Schülerinnen

- ▷ untersuchen bereits implementierte Systeme,
- ▷ kommunizieren mündlich und schriftlich strukturiert über informatische Sachverhalte,
- ▷ tauschen sich mit einem Partner oder in der Kleingruppe über Ergebnisse oder Meinungen aus, die sie beispielsweise durch Internetrecherche gewannen,
- ▷ stellen die so gewonnenen Ergebnisse umgangssprachlich (schriftlich und mündlich) dar,
- ▷ veranschaulichen und beschreiben diese Sachverhalte u. a. mithilfe eines Textes,
- ▷ bereiten ihre Arbeitsergebnisse zur Präsentation auf und stellen diese der gesamten Lerngruppe vor.

Beiträge zum Kompetenzaufbau

An inhaltsbezogenen Kompetenzen werden hauptsächlich Fähigkeiten und Fertigkeiten aus dem Bereich *Algorithmen* (AKBSI, 2008, S.30 ff.) sowie aus *Informatik, Mensch und Gesellschaft* (AKBSI, 2008, S.41 ff.) gefördert. Die Schüler und Schülerinnen

- ▷ verwenden Variablen und Wertzuweisungen,
- ▷ entwerfen, implementieren und beurteilen Algorithmen,
- ▷ erkennen die Notwendigkeit einer verantwortungsvollen Nutzung von Informatiksystemen.

Alle drei Unterrichtsbeispiele beschäftigen sich intensiv mit *Sprache*; in den Bildungsstandards wäre der Bereich *Sprachen und Automaten* zuständig. Leider ließen sich dort keine Kompetenzbeschreibungen auffinden, die der Intention (vor allem der Aufgaben) entsprechen – in künftigen Ergänzungen der Bildungsstandards wäre dies vielleicht zu berücksichtigen.

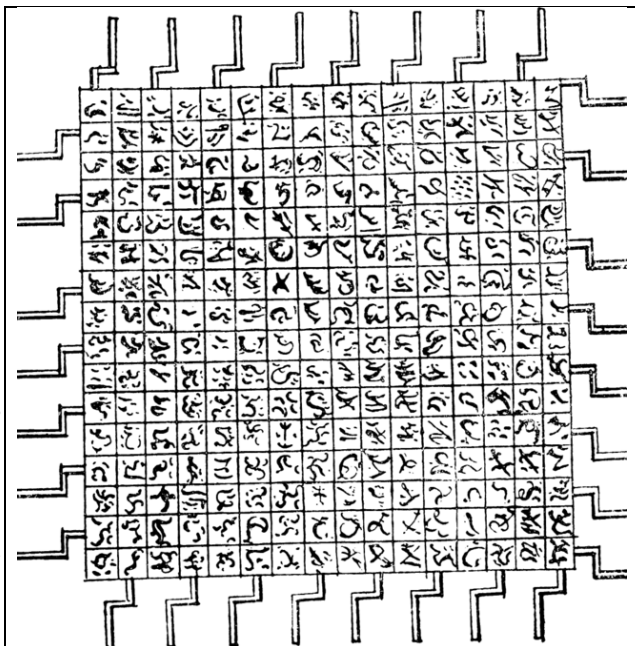
Zufallssätze

Die Sokal-Affäre (siehe Sokal/Bricmont, 1999) wirft ein interessantes Bild auf den Wissenschaftsbetrieb. Der Physiker Alan Sokal hat einen Nonsensartikel aus Begriffen und Versatzstücken der modernen Philosophie geschrieben, der von einer Fachzeitschrift klaglos begutachtet und publiziert wurde (vgl. auch Kasten „Stochastische Texte“, nächste Seite).

Stellen wir uns also die Aufgabe, eine „Phrasendreschmaschine“ zu programmieren, die Zufallssätze erzeugt. Dazu benötigt man zunächst einen Wortschatz. Dieser lässt sich aus einem umfangreichen Text gewinnen (z.B. aus der Sammlung des *Projekts Gutenberg*). Für den Roman *Effi Briest* von Theodor Fontane spricht, dass er häufig im Deutschunterricht gelesen wird. Wünscht man „gute“ Zufallssätze, dürfen die Wörter nicht alle die gleiche Wahrscheinlichkeit besitzen. Es ist nun ganz einfach, die Wahrscheinlichkeiten zu berücksichtigen, indem man den Text in eine Wör-

Stochastische Texte

Anfangs der Sechzigerjahre des vorigen Jahrhunderts, also ganz zu Beginn der Computernutzung in Deutschland, begann man sich, vor allem an der Universität Stuttgart, mit dem Thema *Computergenerierung von Texten* zu beschäftigen. Der Gedanke, den Zufall zum Mit-Produzenten von Texten zu machen, existierte allerdings schon lange, bevor an elektronische Datenverarbeitung auch nur zu denken war. Jonathan Swift berichtet im Jahr 1726 über die phantastischen Reisen seines Titelhelden Gulliver ins Land der Mathematiker. Dort findet er in der großen Akademie von Lagado unter anderem eine mächtige Maschine vor, mit der „der dümmste Mensch ohne Genie und Studien bei ganz geringen Unkosten und mäßiger Leibesbewegung beliebig viele philosophische, politische, juristische, mathematische und theologische Bücher schreiben“ kann.



Quelle: Poetry Foundation, Chicago (USA)

Die „Writing Machine“ in der großen Akademie von Lagado.

Allerdings betrachteten die Benutzer damals nicht den künstlerischen Wert des Produkts, sondern erhofften sich durch die Verwendung kombinatorischer Verfahren bessere Erkenntnisse Gottes und der Welt (Wagner, 1995, S. 11).



Quelle: LOG-IN-Archiv



Rul Gunzenhäuser (links) und Zuse Z22 (rechts, hier im Deutschen Technikmuseum Berlin) schufen das „Weihnachtsgedicht“.

Erste Texte dieser Art wurden von einem Programm erzeugt, das Theo Lutz am Rechenzentrum der TH Stuttgart für eine Zuse Z22 entwickelt hatte; sie wurden von ihm im Jahr 1959 unter der Bezeichnung *Stochastische Texte* veröffentlicht.

Es folgten 1963 Texte eines Programms von Lutz und Rul Gunzenhäuser. Eines dieser Gedichte ist das berühmte Weihnachtsgedicht, das sehr häufig zitiert wurde (siehe Koerber, 1989; Wagner, 1995, S. 12), mit folgendem Schluss:

DER SCHNEE IST KALT
UND JEDER FRIEDE IST TIEF
UND KEIN CHRISTBAUM IST LEISE
ODER JEDE KERZE IST WEIß
ODER EIN FRIEDE IST KALT
UND EIN ENGEL IST REIN
UND JEDER FRIEDE IST STILL
ODER JEDER FRIEDE IST WEIß
ODER DAS KIND IST STILL
EIN ENGEL IST ÜBERALL

„Beinahe interessanter als diese (bei Kenntnis der Entstehungsart recht wenig reizvollen) Wortkombinationen sind die Reaktionen von Lesern, denen die Entstehungsart unbekannt war. R. Gunzenhäuser berichtet von zahlreichen begeisterten Zuschriften ...“, vermerkte Karl Steinbuch (³1965, S. 318), einer der Väter der Informatik in Deutschland.

„Unter der *künstlichen Poesie* wird hier eine Art von Poesie verstanden, in der es, sofern sie z. B. maschinell hervorgebracht wurde, kein personales poetisches Bewusstsein mit seinen Erfahrungen, Erlebnissen, Gefühlen, Erinnerungen, Gedanken, Vorstellungen einer Einbildungskraft etc., also keine präexistente Welt gibt, und in der das Schreiben keine ontologische Fortsetzung mehr ist, durch die der Weltaspekt der Worte auf ein Ich bezogen werden könnte. Infolgedessen ist auch aus der sprachlichen Fixierung dieser Poesie weder ein lyrisches Ich noch eine fiktive epische Welt sinnvoll abhebbar. Während also für die natürliche Poesie ein intentionaler Anfang des Wortprozesses charakteristisch ist, kann es für die künstliche Poesie nur einen materialen Ursprung geben“ (Max Bense, 1962).

Literatur und Internetquellen

Bense, M.: Über natürliche und künstliche Poesie. Köln, 1962.
http://www.stuttgarter-schule.de/natuerliche_und_kuenstliche_poesie.html
[zuletzt geprüft: 30. Dezember 2008]

Gunzenhäuser, R.: Zur Synthese von Texten mit Hilfe programmgesteuerter Ziffernrechenanlagen. In: MTW-Mitteilungen, 10. Jg. (1963), H. 1, S. 4-9.

Koerber, B.: Ein Weihnachtsgedicht. In: LOG IN, 9. Jg. (1989), H. 2, S. 64.

Lutz, Th.: Stochastische Texte.
http://www.netzliteratur.net/lutz_schule.htm
[zuletzt geprüft: 30. Dezember 2008]

Wagner, J.: Computer-Lyrik – oder: Wenn der Blechtrottel dichtet ... In: LOG IN, 15. Jg. (1995), H. 4, S. 10-15.

Schmidt, A.: Computergedichte. In: LOG IN, 10. Jg. (1990), H. 3, S. 33-36.

Steinbuch, K.: Automat und Mensch – Kybernetische Tatsachen und Hypothesen. Berlin u. a.: Springer, ³1965.

terliste verwandelt. Aus „Der Apfel, der am Baum hängt, ist rot“ wird dann (nach Konversion in Klein-schreibung und ohne Satzzeichen) die Liste [„der“, „apfel“, „der“, „am“, „baum“, „hängt“, „ist“, „rot“]. Wählt man einen Index zwischen 0 und 7 zufällig aus, erhält man ein zufälliges Wort, wobei „der“ doppelt so wahrscheinlich ist wie die anderen Wörter. Diese Idee ist in PYTHON mithilfe des Zufallsgenerators `randint(a, b)` leicht zu realisieren:

```
from random import *

text = open("kurz.txt", "r").read() # Text lesen
text = text.lower() # in Kleinbuchstaben verwandeln
text = text.replace("\n", " ").replace("'", '')
text = text.replace(".", " ").replace(",", " ")
text = text.replace(";", " ").replace(":", " ")
text = text.replace(" ", " ")

textlist = text.split(" ")
# In eine Liste verwandeln

def zufallswort():
    global textlist
    i = randint(0, len(textlist)-1)
    return textlist[i]

def zufallssatz(n):
    satz = ""
    for i in range(n):
        satz = satz + " " + zufallswort()
    return satz

print zufallssatz(10)
```

Das Programm (über den LOG-IN-Service zu beziehen) liefert Ergebnisse folgender Art:

- ▷ *wir ich inzwischen aus merkwürdig ich sei das frau kämst*
- ▷ *gar schulter sie auch gefühl drei sie und sie*
- ▷ *strand ich täglichen und das sand es nicht noch nicht*

Damit lässt sich allerdings noch nicht einmal ein dadaistischer oder surrealistischer Dichter imitieren. Die Beispiele zeigen zugleich ein Problem: Die Wortverbindung „das frau“ ist im Deutschen nicht zulässig. Wir sollten Wörter also nicht einfach nach ihrer Häufigkeit überhaupt im Text wählen, sondern nach der Wahrscheinlichkeit, dass ein Wort als Nachfolger eines bestimmten anderen Wortes auftritt. Zu diesem Zweck werden wir zu jedem Wort noch die Liste seiner potentiellen Nachfolger speichern. Mit dem Programm *ZufSätze-02* (siehe LOG-IN-Service) erhält man schon etwas schönere Sätze:

- ▷ *beherbergt und musste dies immer noch den freiplatz neben dem*
- ▷ *innstetten nickte ist ja sie wirft nicht viel besser*
- ▷ *er nachher lies lieber was vom cafe bauer*
- ▷ *sich auf keine lektüre dabei zog verbrachte sie sähen aus*

Allerdings benötigt der Aufbau der Nachfolgerliste viel Zeit. Mit den sogenannten *dictionaries* (Hasch-Tabellen) steht in PYTHON aber eine eingebaute Datenstruktur zur Verfügung, die unter einem Schlüssel (dem Wort) etwas anderes speichert (hier: seine möglichen Nachfolger). Damit kann der mittlere Programmteil

umgeschrieben werden (*ZufSätze-03*). Berücksichtigt man sogar noch den Vorgänger, gelangt man zu Sätzen wie

- ▷ *vor der zeit an einen philosophen wenden oder ein iltis*
- ▷ *in den schluss zu lesen während sie das geschehene vergessen*
- ▷ *dem starken luftzuge der ging und effi trat ein und*
- ▷ *unmöglich für sie hat es nicht seine enkelin was übrigen*

Dafür sind nur kleine Ergänzungen nötig (Programm *ZufSätze-04*).

Aufgabe 1.1: Ein Problem besteht darin, dass wir feste Satzlängen vorgegeben haben. Man sollte in den Vorbereitungen auch nach Wörtern suchen, hinter denen ein Punkt steht, mit denen also ein Satz endet. Und dann bricht man – wieder zufallsgesteuert – die Saterzeugung ab, wenn ein solches wahrscheinliches Schlusswort erzeugt ist. Ergänzen Sie das Programm in diesem Sinn.

Aufgabe 1.2: Schreiben Sie ein Programm nach dem Muster des Weihnachtsgedichts (siehe Kasten „Stochastische Texte“). Verwenden Sie dazu (z. B. nach dem Roman *Das Schloss* von Franz Kafka) 16 Substantive, etwa DER GRAF, DER FREMDE, DER BLICK, DIE KIRCHE, DAS SCHLOSS, DAS BILD, DAS AUGEN, DAS DORF, DER TURM, DER BAUER, DER WEG, DER GAST, DER TAG, DAS HAUS, DER TISCH, DER KNECHT und 16 Adjektive, etwa OFFEN, STILL, STARK, GUT, SCHMAL, NAH, NEU, LEISE, FERN, TIEF, SPÄT, DUNKEL, FREI, GROSS, ALT, WÜTEND. Alle Substantive bzw. Adjektive sollen gleich häufig auftreten. Die beiden Elementarsätze eines Paares sollen durch logische Konstanten verknüpft werden, und zwar durch UND mit einer relativen Häufigkeit von 1/8, durch ODER (1/8), durch SO GILT (1/8) und durch einen Punkt (5/8). Als logische Operatoren sollen mit gleicher Häufigkeit verwendet werden: der Partikularisator EIN, EINE, EIN; der Generalisator JEDER, JEDE, JEDES; der verneinte Partikularisator KEIN, KEINE, KEINES und der verneinte Generalisator NICHT JEDER, NICHT JEDE, NICHT JEDES.

Aufgabe 1.3: Erzeugen Sie Computer-Gedichte nach dem Muster von Gerhard Stickels *Autopoem 312* (IBM 7090, 1967):

DIE FRÖHLICHEN TRÄUME REGNEN
 DAS HERZ KÜBT DEN GRASHALM
 DAS GRÜN VERSTREUT DEN SCHLANKEN GELIEBTEN
 FERN IST EINE WEITE UND MELANCHOLISCH
 DIE FÜCHSE SCHLAFEN RUHIG
 DER TRAUM STREICHELTE DIE LICHTER
 TRAUMHAFTES SCHLAFEN GEWINNT DIE ERDE
 ANMUT FRIERT, WO DIESES LEUCHTEN TÄNDELT
 MAGISCH TANZT DER SCHWACHE HIRTE

Diskutieren Sie die Aussagen *Max Benses* über künstliche Poesie (siehe Kasten „Stochastische Texte“).

Aufgabe 1.4: Nehmen Sie zu folgender These Stellung!

„Nonsense hat vielleicht den Vorteil, dass er unserem Denken neue Möglichkeiten erschließt. Die bloße Nebeneinanderstellung einiger beliebiger Wörter kann den Geist dermaßen beflügeln, dass er in imaginäre Welten aufsteigt. Gleichsam, als sei das, was Sinn hat, zu irdisch, zu weltlich, und als hätten wir zwischendurch eine Verschnaufpause nötig. Vielleicht engt Sinnhaltiges auch zu sehr ein. Nonsense rückt die unverständliche Seite des Universums in den Vordergrund, Sinnreiches dagegen die fassbare“ (Hofstadter, 1988, S. 232).

Aufgabe 1.5: Bestätigen oder widerlegen Sie (an Beispielen) folgende Aussage:

„Der Computer fügt wahl- und intentionslos zusammen, er produziert Unsinn. [...] Der Text muss also vom jeweiligen Rezipienten individuell und in einem konstruktiven Akt mit Sinn erfüllt werden. Der Sinn eines computer-generierten Textes wird deshalb auch nicht – wie manchmal behauptet – vom Zufall bestimmt, sondern allein vom Leser“ (Wagner, 1995, S. 13).

Rechtschreibkorrektur

Peter Norvig (Bild 1) ist ein bekannter amerikanischer Informatiker und KI-Wissenschaftler, derzeit Leiter der Forschungsabteilung bei *Google*. In dem Artikel *How to Write a Spelling Corrector* hat er mit einem erstaunlich kurzen PYTHON-Programm demonstriert, wie man mit Techniken, die wir soeben kennengelernt haben, eine Rechtschreibkorrektur durchführen kann, und wie sie etwa Google anbietet.

Aufgabe 2.1: Experimentieren Sie mit der Rechtschreibkorrektur bei *Google* (und anderen Suchmaschinen ihrer Wahl), indem Sie typische Beispiele für erfolgreiche Korrektur bzw. fehlerhaftes Verhalten auffinden und notieren. Was lässt sich daraus über die Arbeitsweise des jeweiligen Systems vermuten?



Bild 1: Peter Norvig, Forschungsdirektor bei Google.

Quelle: flickr

Wenn man als Suchwort „Apfell“ eingibt, wird man sogleich gefragt, „Meinten Sie: Apfel?“ Norvig stellte folgende Überlegungen an:

- Man braucht einen großen Text, um zu wissen, welche Wörter es in einer Sprache überhaupt gibt, und welche davon wie häufig sind.
- Tippfehler sind meistens „kleine“ Veränderungen des richtigen Wortes: Ein vergessener Buchstabe, ein Buchstabendreher, ein zufällig hinzugefügter Buchstabe.
- Wenn man eine Eingabe hat, die vermutlich einen Tippfehler darstellt, kann man nach (b) auflisten, welche Wörter wohl gemeint sein könnten, und unter dieser Menge kann man ein wahrscheinliches, also häufiges Wort gemäß (a) auswählen.

Das Programm *Rechtschreib* lehnt sich an Norvigs Programm an; es ist nicht ganz so kompakt, dafür aber verständlicher. Die Analyse nach Teil (a) wird durch Auswertung eines großen Beispieltextes durchgeführt. Dazu wird in einem Wörterbuch (dictionary) die Häufigkeit der Wörter gespeichert. Die unter (b) beschriebenen Fehler beim Tippen eines Wortes stellt die Funktion `edits1` zusammen. Der folgende Aufruf von `edits1` listet die Wörter auf, von denen „Apfell“ ein Vertipper sein könnte:

```
>>> edits1("Apfell")
['pfell', 'Afell', 'Apell', 'Apfl', 'Apfel', 'Apfel', 'pApell', 'Apfell', 'Apefl', 'Apflel', 'Apfell', 'Apfeal', 'Apfebl', 'Apfecl', 'Apfedl', 'Apfeel', 'Apfefl', 'Apfegl', 'Apfehl', ...]
```

Unter diesen Möglichkeiten werden nur die betrachtet, die in der Textbasis vorkommen, also als bekannte Wörter gelten; das prüft die Funktion `known`. Die Korrekturfunktion `correct` zeigt erstaunliche Leistungen, aber auch Fehlleistungen:

```
>>> correct("Libe")
'liebe'
>>> correct("Libbe")
'liebe'
>>> correct("Lippe")
'lippe'
>>> correct("Kopff")
'kopf'
>>> correct("meer")
'er'
```

Als Textbasis wurde wieder *Effi Briest* verwendet, die allerdings zu klein ist: Weder Internet noch Äpfel noch das Meer sind bekannt. Das Schöne an Norvigs Korrektor ist, dass er mit seiner großen Textbasis wirklich „industrial strength“ besitzt.

Aufgabe 2.2: Verbessern Sie das Programm, indem Sie Wörtern, die nur durch einen Tippfehler (`edits1`) entstehen, gegenüber solchen, die aus zweien entstehen, einen Vorsprung in der Wahrscheinlichkeitswertung geben.

Aufgabe 2.3: Studieren Sie Norvigs Original-Programm und vergleichen Sie!

Eliza

Im Jahr 1966 entwickelte der deutsch-amerikanische Computerwissenschaftler Joseph Weizenbaum (1923–2008, siehe LOG IN, H. 150/151, S.8) am MIT in Cambridge (Mass.) ein Programm, mit dem sich eine Art Unterhaltung führen ließ. Der menschliche Gesprächspartner tippte seine Äußerung auf einer mit dem Computer verbundenen Schreibmaschine ein; der Computer analysierte diese und stellte in natürlicher Sprache (Englisch) eine Antwort zusammen, die wiederum über die Schreibmaschine ausgedruckt wurde (Bildschirme gab es damals noch nicht). Das Programm war als Zwei-Bänder-Anordnung konstruiert, wobei das erste Band den *Sprachanalytiker* und das zweite das sogenannte *Skript* enthielt. Ein Skript besteht aus einer Anzahl von Regeln, die etwa denen gleichen, an die ein Schauspieler gebunden ist, der über ein bestimmtes Thema improvisiert. Somit konnte man *Eliza* ein Skript vorgeben, das sie in die Lage versetzte, ein Gespräch z. B. über das Kochen von Eiern oder die Benutzung eines Girokontos auf der Bank usw. zu führen. Das heißt: Jedes Skript erlaubte *Eliza* die Übernahme einer spezifischen Gesprächsrolle.

Weizenbaum wählte ein Skript, das *Eliza* die Rolle eines Psychiaters spielen ließ, der mit einem Patienten ein erstes Gespräch führt. Ein solcher Therapeut ist nach Weizenbaum vergleichsweise leicht zu imitieren (oder besser: zu parodieren), da ein Großteil seiner Gesprächstechnik darin besteht, den Patienten zum Sprechen zu bringen, indem er diesem seine eigenen Äußerungen als eine Art Echo zurückgibt.

Das Programm wirkte auf seine Benutzer so mitfühlend und verständlich, dass es tatsächlich eine Art therapeutischer Funktion bekam. Personen, die sich mit *Eliza* unterhielten bauten rasch eine emotionale Beziehung zum Computer auf und schrieben ihm eindeutig menschliche Eigenschaften zu (siehe auch Witten/Horning, 2008, Seite 51–60 in diesem Heft).

Aufgabe 3.1: Suchen Sie im Internet eines der zahlreichen *Eliza*-Programme und führen Sie ein „Gespräch“ mit ihm. Wie kommt die Wirkung des scheinbaren Verstehens bzw. Mitfühlens zustande? Worin besteht die „Gesprächstechnik“ des Programms?

Ein *Muster* (oder: *Schablone*) ist ein Term, in dem bestimmte Stellen als „beliebig“ markiert sind. Wir können auch sagen, dass das Muster *Leerstellen* (Löcher, engl.: *slots*) enthält, die sich füllen lassen. Ein Muster steht damit für eine ganze Klasse von Termen, nämlich für alle, auf die das Muster *passt*, d. h. die von der Form des Musters sind. Die PYTHON-Implementation verwendet folgende Satz- und Antwortschablonen:

```
[["ich", "bin", "_a"],
 ["warum", "bist", "du", "_a"]],
[["ich", "habe", "_a"],
 ["hast", "du", "_a", "schon", "laenger?"]],
[["ich", "bin", "_a"],
 ["warum", "bist", "du", "_a", "?"]],
```

```
[["das", "ist", "_a"],
 ["warum", "findest", "du", "dass", "das",
 "_a", "ist?"]]
```

Dabei ist eine Zeichenkette, die mit einem Unterstrich anfängt, eine Variable für ein Wort, und eine Variable, die mit zwei Unterstrichen anfängt, eine Variable, die für beliebig viele Zeichen stehen kann. Schwierig ist hier vor allem die Funktion `match()`, die prüft, ob ein Satz auf ein Muster passt. Sie lautet:

```
def match(satz, muster):
    # Gibt False oder Liste von Substitutionen
    if satz == [] and muster == []: return []
    if satz == [] or muster == []: return False
    if satz[0] == muster[0]: return match(satz[1:],
        muster[1:])
    if not(istVar(muster[0])): return False
    if not(istMVar(muster[0])): # Normale Variable
        a = match(satz[1:], muster[1:])
        if a == False: return a
        a.append([muster[0], [satz[0]]])
        return a
    var = muster[0] # Multi-Variable
    if len(muster) == 1: return [[var, satz]]
    naechstes = muster[1]
    varwert = []
    i = 0
    for w in satz:
        if w == naechstes: break
        varwert.append(w)
        i = i + 1
    a = match(satz[i:], muster[1:])
    if a == False: return a
    a.append([var, varwert])
    return a
```

Das folgende Beispiel zeigt, wie `match()` funktioniert:

```
print match(["das", "ist", "bello", "und", "er",
 "bellt"],
 ["das", "ist", "_hund", "und", "er",
 "_tut"])
[['_tut', ['bellt']], ['_hund', ['bello']]]
```

Die anderen Dinge sind nicht sehr aufregende Algorithmen. Daran anschließend könnte man im Unterricht



Quelle: LOG-IN-Archiv

Bild 2: George Bernard Shaw (1856–1950) auf dem Titelblatt des Magazins *TIME* im Jahr 1923 und *Pygmalion* („Pygmalion und Galatea“ von Jean-Léon Gérôme, Öl auf Leinwand, 1890).

reguläre Ausdrücke, Produktionssysteme oder logisches Programmieren thematisieren.

Aufgabe 3.2: Erweitern Sie das Programm um neue Muster. Dabei werden Sie u. U. feststellen, dass noch Prüfungen einzubauen sind: Es mag sinnvoll sein, auf „Ich liebe X“ zu antworten „Glaubst Du, dass X dich auch liebt?“ – aber nicht, wenn X für „Pizza“ steht.

Aufgabe 3.3:

- (a) Recherchieren Sie im Internet über Weizenbaums Beweggründe, sein Programm „Eliza“ zu nennen (Stichwörter: G.B. Shaw, Pygmalion; siehe Bild 2, vorige Seite).
- (b) Studieren Sie Weizenbaums Original-Arbeit (<http://i5.nyu.edu/~mm64/x52.9265/january1966.html>).

Aufgabe 3.4: Führen Sie je einen Dialog mit dem PYTHON-Programm – und zwar so, dass

- (a) die Illusion genährt wird, beide Dialogpartner seien Menschen,
- (b) erkennbar ist, dass einer kein Mensch sein kann. Worin besteht der Unterschied?

Aufgabe 3.5: Einer der frühesten Versuche, im Netz (um 1985 noch *Usenet* genannt) einen intelligente Dialogpartner zu imitieren, stammt von Bruce Ellis und trug den Namen „Mark V. Shaney“ (Anklang an *Markov chaining*, s. u.). Es griff die „postings“ anderer Netzteilnehmer auf, zerlegte sie und arrangierte sie neu, indem es Markoff-Ketten (Wahrscheinlichkeiten für Wortübergänge, ähnlich wie im obigen PYTHON-Programm *ZufSätze*) verwendete (siehe Dewdney, 1989). Recherchieren Sie im Internet bezüglich *Mark V. Shaney* und vergleichen Sie mit *Eliza*.

Aufgabe 3.6: Diskutieren Sie A.K. Dewdney's These: „Semantischen Unsinn akzeptiert der Mensch eher als syntaktische Schnitzer.“

Aufgabe 3.7: Nehmen Sie Stellung!

„Vor meiner Wandlung habe ich mit Lust und Eifer das Feld der Computerwissenschaft bebaut. Nicht etwa deshalb, weil es besonders einträglich gewesen wäre – mitnichten. Wenn man so mit Leidenschaft arbeitet, ist der Grund immer der, dass die Arbeit Vergnügen bereitet: etwa wie Rätsellösen. R. Oppenheimer wurde einmal gefragt, warum er und seine Leute mit solchem Enthusiasmus an der Atombombe habe arbeiten können. It was sweet science, sagte er ...“ (Joseph Weizenbaum, in: *Kurs auf den Eisberg*, 1987).

Kreationismus und Evolution im Computer

Kehren wir noch einmal zum Ausgangspunkt, nämlich zur maschinellen Erzeugung möglichst eleganter (aber leider sinnloser) Sätze zurück. Dieser scheinbar abwegigen Tätigkeit hat sich auch David H. Bailey (Universität Berkeley) zugewandt.

Bailey ist ein anerkannter guter Mathematiker, der – zusammen mit Peter Borwein und Simon Plouffe – ein Verfahren entwickelt hat, das die Berechnung von Sedezimalziffern der Kreiszahl π ermöglicht, ohne die vorhergehenden Stellen zu kennen (siehe Arndt/Hanel, 1998, S. 87 ff.). Außerdem hat er einen Algorithmus gefunden, mit dem man effizient ganzzahlige Relationen zwischen reellen Zahlen bestimmen kann: eine verblüffende Erweiterung des Euklidischen Algorithmus.

Wie kommt ein so intelligenter Mensch dazu, unsinnige Sätze zu erzeugen? In den USA haben Evolutionskritiker wie die Kreationisten und die Vertreter des „Intelligent Design“ erheblichen Einfluss. Zu den üblichen Argumenten gegen die Evolutionstheorie gehört, dass blinde Auslese so komplexe Dinge wie Lebewesen oder literarische Texte nicht hervorbringen könne.

Einer der ersten Kreationisten war John Tillotson (1630–1694), Erzbischof von Canterbury, der als Argument für den göttlichen Ursprung der Schöpfung Folgendes schreibt: „Wie oft müsste wohl ein Mensch, der die Buchstaben des Alphabets in einem Sack gut durchgeschüttelt hat, sie auf den Boden austreuen, ehe sie ein

Foto: University of California, Berkeley

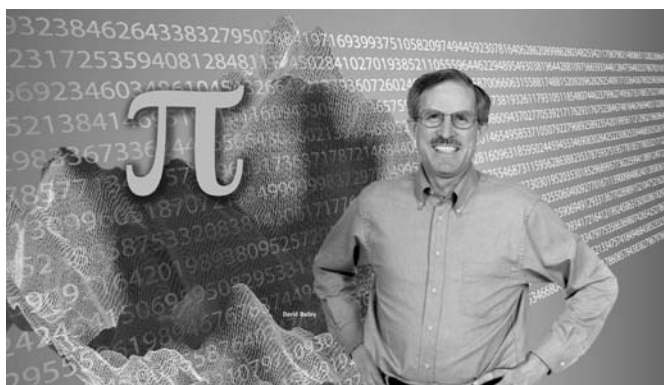


Bild 3: David H. Bailey vor 29 360 999 Dezimalziffern von π .



Bild 4: John Tillotson (1630–1694), Erzbischof von Canterbury, einer der Urväter des Kreationismus.

http://www.piburns.com/jfbisno2.jpg

komplettes Gedicht, ja auch nur einen guten Gedanken in Prosa ergäben? Und könnte das große Buch der Schöpfung etwas leichter durch Zufall entstehen als ein kleiner Prosaband?“

Also hat Bailey sich nun – zusammen mit dem Supercomputer des *Lawrence Berkeley National Laboratory* – daran gemacht, durch evolutionäre Prozesse englische Sätze zu erzeugen und damit Tillotson zu widerlegen. (Zur „Evolution im Computer“ siehe Berchtold, 1990). Baileys Grundideen sind:

Erstens: Es wird ein umfangreicher Text (bei ihm Charles Dickens: *Great Expectations*) als Textbasis verwendet, aus dem zufällig Buchstaben gezogen werden. Das Programm verbessert ständig eine Menge von Zufallsätzen, die als Population bezeichnet werden, und die anfangs Buchstabe für Buchstabe zufällig aus der Textbasis gezogen werden (Programm Evolution-01). Die Population besteht dann aus einer Liste von 512 Zeichensequenzen, die alle die gleiche Länge haben. In diesen Zeichenketten kommen auch Sonderzeichen vor, etwa:

- ▷ *o\necef sae tdid eweketee?nnsec;"z negs\nd atvii oineosnemag. Eeu*
- ▷ *dhk wss duo elp dsuia aiinte r lkunoeeba nr ihl e\innauewabtrd" c*
- ▷ *ieihnwee.n e.ieeeo eioeisg is aoaintonvenewee\vnnaad,satwdhi fswa*
- ▷ *li,fdas;hg be ,eewnu\enneh,earbkn\nb" islh nshgulwacerhkhineikirh*

Zweitens: Aus dieser Population, d.h. Menge von Zeichenketten, werden jetzt durch evolutionäre Prozesse noch viel mehr. Diese Prozesse sind:

- ▷ Kombination: Zwei Zeichenketten tauschen eine zufällige, 4 Zeichen lange, Teilkette aus (Funktion *mate*).
- ▷ Ein Zeichen an einer zufälligen Position wird durch ein anderes zufälliges Zeichen ersetzt (Funktion *mut1*).
- ▷ Zwei aufeinanderfolgende Zeichen werden durch zwei zufällige ersetzt (Funktion *mut2*).
- ▷ In den Text wird an einer zufälligen Stelle ein zufälliges Zeichen eingefügt. Weil dadurch zunächst eine Zeichenkette der Länge 65 entsteht, wird das letzte Zeichen weggelassen (Funktion *mut3*).
- ▷ Aus der Zeichenkette wird an einer zufälligen Position ein Zeichen gelöscht, und zum Ausgleich hinten ein zufälliges Zeichen angefügt (Funktion *mut4*).

Diese Mutationen (Programm Evolution-02) ähneln in manchem Norvigs Rechtschreibkorrektur!

Drittens: Die Population wird jetzt stark vergrößert, die Zufallssequenzen bekommen „Nachwuchs“ und zwar wie folgt:

- ▷ Die ersten 20 werden miteinander kombiniert und erzeugen so 400 neue Zeichenketten (Funktion *paarungen1()*).
- ▷ Die ersten 100 werden außerdem zufällig mit zufälligen anderen aus der gesamten Population kombiniert (Funktion *paarungen2()*).
- ▷ Die vier Mutationsfunktionen werden auf 400, 200, 100 bzw. 100 zufällig ausgewählte Zeichenketten an-

gewendet. Die zufällige Auswahl von Zeichenketten aus der Population erledigt die Funktion *gen*.

Diese Operationen zusammen bilden eine sogenannte *Runde* der Evolution (Programm Evolution-03). Gemäß der Idee „Überleben des Fittesten“ wird die Fitness der Zeichenketten bewertet, und nur die 512 (ips) fittesten „überleben“, alle anderen werden vergessen.

Viertens: Fitness beurteilen. In diesem Schritt liegt die Zielsteuerung. Hier entscheidet sich, ob aus dem Buchstabensalat englische oder deutsche Sätze entstehen. Um die Fitness einer Zeichenkette *s* zu bewerten wird nach Bailey folgendermaßen vorgegangen:

- ▷ Es wird geprüft, ob *s[0:16]* oder *s[1:17]* oder eine andere Teilkette der Länge 16 in der Textbasis vorkommt. In diesem Falle erhöht sich der Fitnesswert um 16.
- ▷ Es wird geprüft, ob *s[0:15]* oder *s[1:16]* oder eine andere Teilkette der Länge 15 in der Textbasis vorkommt. In diesem Falle erhöht sich der Fitnesswert um 15; usw.

Zu diesem Zweck speichert man beim Einlesen 16 dictionaries in der Liste *sgs*, wobei *sgs[i]* ein dictionary ist, das alle Teilzeichenketten aus der Textbasis der Länge *i* enthält (Programm Evolution-04).

In jeder Runde wird nun die neue, größer gewordene Population gemäß dieser Fitnessfunktion sortiert, sodass die fittesten Zeichenketten immer am Anfang der Population stehen. Schließlich muss man den Prozess nur noch hinreichend viele Runden (hier 2500) lang seine Arbeit tun lassen und dann die beste Sequenz ausgeben:

```
def gen(): # erzeuge einen Zufallssatz
    global population, ips, L, text
    population = ["" ] * ips
    for i in range(ips):
        population[i] = zufallstext(L)
        counter = 250 # 2500 für bessere Ergebnisse
        while counter > 0:
            counter = counter - 1
            eineRunde()
        return population[0]
for i in range(5): print(gen())
```

Eine Ausgabe des Programms (nach 2500 Runden) lautet beispielsweise:

- ▷ *st also alles als eine alles als einen reinen einer an einen kin*
- ▷ *einen seiner kleinen kleinen leine das seiner kleinen leine das*

Dieses Ergebnis lässt sich mit etwas mehr Geduld (mehr Runden) noch verbessern. Bailey hat Studenten, bunt gemischt, Original-Dickens-Ausschnitte und Zufallsprodukte vorgelegt. Die Studenten konnten im Schnitt nur bei 61 % korrekt den Urheber – Dickens oder die Maschine – ermitteln. Einige seiner Maschinensätze:

- ▷ *fitted it to nothing and get the ashes between me to the last*
- ▷ *as no relation into another that it is the same room - a little*

▷ *a separation to be made for the desolater, like the man he was*

Zum Vergleich einige echte Dickens-Sätze:

▷ *he saw me going to ask him anything, he looked at me with his glass*

▷ *on my objecting to this retreat, he took us into another room with*

▷ *the chimney as though it could not bear to go out into such a night*

Eine naheliegende Kritik könnte sein, dass das Programm schließlich nur bei Sätzen landet, die in der Textbasis bereits vorkommen. Das ist nicht der Fall. Bailey hat deutlich mehr Zufallstext erzeugt, als die Dickens-Vorlage enthält. Interessanterweise erfindet das Programm auch viele Wörter, die in der Textbasis gar nicht vorkommen, die es aber wirklich gibt, oder aber die zumindest vernünftig auszusprechen sind.

Aufgabe 4.1: Bauen Sie noch folgende Mutationsvarianten ein, die es in Dann-Sequenzen gibt: Zwei Ketten werden geteilt ($a = a_1 + a_2$, $b = b_1 + b_2$) und zwei neue Ketten resultieren daraus: $a_1 + b_2$, $b_1 + a_2$.

Ein solches Programm in PYTHON zu schreiben, hat Vor- und Nachteile: Bailey hat in FORTRAN programmiert und dafür 1000 Zeilen Programmtext schreiben müssen. Das zeigt, wie viel ausdrucksstärker PYTHON ist. Für die Schule ist das wichtiger als der Nachteil geringerer Effizienz des PYTHON- gegenüber dem FORTRAN-Programm.

Aufgabe 4.2: Es ist klar, dass sich gläubige Kreationisten von Computerspielereien wie dieser nicht beeindruckt lassen. Informieren Sie sich im Internet über die Argumente des Kreationismus und nehmen Sie Stellung!

Ausblick

Eines der spannendsten Themen in diesem Bereich wurde noch gar nicht berührt, nämlich die automatische Übersetzung auf der Grundlage großer Texte, die in zwei Sprachen vorliegen. Über *Google* wurde dieser Ansatz weit verbreitet. Obwohl die Fehler der Google-Übersetzungsmaschine manchmal so eigenartig sind, dass sie schon literarisch verarbeitet wurden (Brammertz, 2006), muss man der Leistung auch Respekt zollen. Eine didaktische Reduktion dieser Verfahren steht aber noch aus.

Prof. Dr. Reinhard Oldenburg
Institut für Didaktik der Mathematik und Informatik
Senckenberganlage 9
60325 Frankfurt

E-Mail: oldenbur@math.uni-frankfurt.de

Alle besprochenen PYTHON-Programme können über den **LOG-IN-Service** (siehe S. 128) bezogen werden.

Literatur und Internetquellen

Arndt, J.; Haenel, Chr.: *Pi – Algorithmen, Computer, Arithmetik*. Berlin u. a.: Springer, 1998.

Bailey, D. H.: *Can An Evolutionary Process Create English Text?*
<http://crd.lbl.gov/~dhbailey/dhbpapers/dhb-english-text.pdf>

Bayer, K.: *Computerlinguistik im Unterricht – Reflexion über Sprache anlässlich der Entwicklung eines „sprechenden Automaten“*. In: LOG IN, 12. Jg. (1992), H. 1, S. 39–49.

Berchtold, St.: *Evolution im Computer – Optimierung mit genetischen Algorithmen*. In: LOG IN, 10. Jg. (1990), H. 1, S. 35–39.

Brammertz, U.: *God save the Queen – Gott speichert die Königin. Irrsinnige und absurde Übersetzungen aus dem Internet*. Hamburg: cadeau (Hoffman und Campe), 2006.

Dewdney, A. K.: *Computer Recreations – A potpourri of programmed prose and prosody*. In: *Scientific American*, Vol. 260 (1989), H. 6, S. 122–125 [deutsch: *Der hoh(1)e Literaturprozessor*. In: Diemer, I. (Hrsg.): *Computer-Kurzweil 2*. Heidelberg: Spektrum, 1990, S. 119–114].

Gutenberg-Projekt:
<http://www.gutenberg.org/>
oder:
<http://gutenberg.spiegel.de/>

Hayes, B.: *Computer Recreations – A progress report on the fine art of turning literature into drivel*. In: *Scientific American*, Vol. 249 (1983), H. 11, S. 18–28 [deutsch: *Computer-Dichtkunst*. In: Diemer, I. (Hrsg.): *Computer-Kurzweil*. Heidelberg: Spektrum, 1988, S. 218–224].

Hofstadter, D. R.: *Metamagicum – Fragen nach der Essenz von Geist und Struktur*. Stuttgart: Klett-Cotta, 1988.

Sokal, A.; Bricmont, J.: *Eleganter Unsinn*. München: dtv, 1999.

Norvig, P.: *How to Write a Spelling Corrector*.
<http://norvig.com/spell-correct.html>

Sahr, A.: *Maschinelle Sprachverarbeitung durch menschliche Intelligenz*. In: LOG IN, 19. Jg. (1989), H. 2, S. 11–21.

Weizenbaum, J.: *Die Macht der Computer und die Ohnmacht der Vernunft*. Frankfurt a. M.: Suhrkamp, 1977.

Witten, H.; Hornung, M.: *Chatbots – Teil 1: Einführung in eine Unterrichtsreihe zu „Informatik im Kontext“ (IniK)*. In: LOG IN, 28. Jg. (2008), H. 154/155, S. 51–60 (*in diesem Heft*).

Alle Internetquellen wurden zuletzt am 30. Dezember 2008 geprüft.