

Universität Hamburg
Department Informatik
Knowledge Technology, WTM

Generieren dynamischer Antworten durch Parsing von natürlicher Sprache

Proseminararbeit

Proseminar: Künstliche Intelligenz

Louis Kobras

Matr.Nr. 6658699

4kobras@informatik.uni-hamburg.de

7. Juli 2015

Abstract

Seit Jahrzehnten schon beschäftigen sich Forscher damit, den Computer dem Menschen näher zu bringen. Ein wichtiger Bestandteil dieses Prozesses ist dabei die Sprachverarbeitung.

Ziel dieser Arbeit ist, Licht auf den Prozess des Verständnisses eines Computers von natürlicher Sprache zu werfen.

Zu diesem Zweck wird der Begriff der natürlichen Sprache in seine Komponenten zerlegt, die einzeln analysiert und in dieser Form von endlichen, deterministischen Automaten verarbeitet werden können.

Im Hinblick auf die Entwicklung von Videospielen wird mithilfe der *String*-Klasse von Java, der *Regex*-Konvention und dem ELIZA-Programm eine Möglichkeit gezeigt, jeden beliebigen Eingabetext zu analysieren und darauf zu reagieren.

Inhaltsverzeichnis

1	Einführung	2
1.1	Was bisher geschah: Sprachverarbeitung im Kontext der historischen Artificial Intelligence	2
1.2	ELIZA als Beispiel für frühe Sprachverarbeitung	2
2	Was ist natürliche Sprache?	3
3	Parsing der Morphologie mithilfe von deterministischen endlichen Automaten	4
4	Regex zur Vereinfachung der Parsing-Arbeit	6
5	Die Arbeitsweise von ELIZA	8
6	Praktischer Nutzen der Textverarbeitung in Hinblick auf Spieleentwicklung	9
7	Schlussfolgerung	10
	Quellenverzeichnis	12
	Anhang 1: Grafiken	I

1 Einführung

Schon seit etwa achtzig Jahren beschäftigen sich Forscher mit dem Thema der Künstlichen Intelligenz, und nicht weniger Zeit ist in die Theorie der Sprachverarbeitung investiert worden. Sprachverarbeitung liegt in zwei Formen vor: Beide Aspekte fallen unter das *Verarbeiten natürlicher Sprache* (Natural Language Processing). Diese Arbeit wird sich mit dem Verarbeiten natürlicher Sprache in Textform beschäftigen.

Zwar ist Kommunikation durch Texteingabe nicht so schnell und dynamisch anwendbar wie gesprochene Anweisungen, dennoch aber ist sie ein wichtiger Schritt dorthin. Wir werden sehen, wie natürliche Sprache aufgebaut ist, wie sie durch Verwendung von Deterministischen Automaten und Regulären Ausdrücken analysiert werden kann, und wie mit den erhaltenen Informationen verfahren werden kann.

1.1 Was bisher geschah: Sprachverarbeitung im Kontext der historischen Artificial Intelligence

Die Grundsteine für Künstliche Intelligenz wurden 1936 von Turing und darauf aufbauend in den 40ern und 50ern gelegt (McCulloch und Pitts (1943), Chomsky (1956) et. al.).

Ende der 50er Jahre hatten sich daraus zwei parallele Zweige für die Sprachverarbeitung gebildet: Das formale Paradigma und der stochastische Ansatz ([4]).

Während das formale Paradigma seinen Schwerpunkt auf formale Sprachen und das Parsen von Schlüsselwörtern gelegt hatte, lag der Schwerpunkt des anderen Ansatzes auf der Berechnung der Wahrscheinlichkeit des Auftretens bestimmter Wörter anhand eines umfassenden Lexikons und der Anwendung Bayesscher Methoden auf die einzelnen Zeichen.

Der stochastische Zweig blieb weitestgehend erhalten und begann, versteckte Markov-Modelle (HMM) bei seiner Analyse zu nutzen. Im Gegensatz dazu teilte sich der formale Zweig in drei Unterpfade. Ein Logik-basierender Zweig befasste sich mit der Vereinheitlichung der Struktur von Sprache und nutzte dabei eine sog. *Definite Clause Grammar*, woraus sich später Prolog entwickeln würde. Der Zweig, der sich mit natürlicher Sprache befasste, versuchte, Text-Kommandos und Konzepte so umzusetzen, dass Maschinen eine semantische Grundlage mitgegeben werden konnte, auf der die Interpretation aufbauen würde. Der Diskurs-Zweig verwendete die Betrachtung von Substrukturen im Diskurs von Sprachen und der Automatisierung von Referenzen.

Die 80er und 90er Jahre beschäftigten sich Wahrscheinlichkeitsmodellen und Modellen, die sich deterministische, endliche Automaten zur Analyse zu Nutzen machten.

Ende des 20. Jahrhunderts hatte sich das Wahrscheinlichkeitsmodell als Standard durchgesetzt. Die Forschung in diesem Feld wurde durch das Aufkommen des Internet und auf kommerzieller Basis durch Entwicklung der Technologie weiter vorangetrieben.

1.2 ELIZA als Beispiel für frühe Sprachverarbeitung

ELIZA ist ein frühes Programm zur Sprachverarbeitung (Weizenbaum, 1966). Das Programm wendet simples Parsing und Mustererkennung an, um Schlüsselwörter aus ei-

nem gegebenen Satz zu extrahieren und darauf aufbauend eine Antwort zu generieren. ELIZA arbeitet als Rogativer Psychologe, d.h. man erzählt ihr etwas und sie stellt eine darauf basierende Frage.

Das Programm ist sehr simpel, und ELIZA benötigt fast keine Informationen, um zu funktionieren. Durch ihre Arbeitsweise kann sie ein natürliches Gespräch mit einem Psychologen nur durch das Wiederholen von Sätzen in Frageform erfolgreich simulieren. Dies tut sie, indem sie mit einem simplen Such-Algorithmus den vom Benutzer eingegebenen Text mit einer Liste von Stichworten vergleicht und anhand derer eine vordefinierte Antwort zurückliefert. Zugegebenermaßen ist die Art der Antworten derart gut konstruiert, dass viele der Probanden, die in den 60er Jahren ersten Kontakt mit ELIZA hatten, der Ansicht waren, sie wäre ein denkendes Programm, fähig zu höherer Intelligenz, und glaubten selbst dann noch an ihre Fähigkeiten, als man ihnen ihre Funktionsweise offenbarte¹.

Dennoch ist ihre künstliche Intelligenz in keiner Weise mit tatsächlicher künstlicher Intelligenz zu vergleichen. Während ELIZA ein erweitertes Key-Value-Pair matching-Verfahren (vgl. 5) anwendet (sie sucht sich eine Antwort anhand eines erkannten Stichwortes), wird von einer KI erwartet, tatsächlich zu verstehen, was zu ihr gesagt wird. Sie muss also nicht nur ein Wort wiedererkennen, sondern auch wissen, was damit gemeint ist. Desweiteren muss eine KI auch in der Lage sein, Wörter in ihrer flexierten Form wiederzuerkennen, eingebaut in natürliche Sprache. Die genaue Arbeitsweise von ELIZA sowie eine Erweiterung dieser zum Erkennen flexierter Wörter wird in dieser Arbeit beleuchtet werden.

2 Was ist natürliche Sprache?

„By 'natural language' we mean a language that is used for everyday communication by humans; languages such as English, Hindi, or Portuguese. In contrast to artificial languages such as programming languages and mathematical notations, natural languages have evolved as they pass from generation to generation. and are hard to pin down with explicit rules.“ ([2], S. ix)

So definieren Steven Bird et. al. den Begriff *natürliche Sprache*. Eine Sprache ist eine Konvention zur Kommunikation bestehend aus einer Grammatik, einem Vokabular und einem Verwendungskontext.

Eine natürliche Sprache ist eine Sprache, die sich im Verlauf der Zeit aus der Kommunikation zwischen Menschen entwickelt hat. Durch die fortwährende Entwicklung einer Sprache verändert sich permanent die Bedeutung und Verwendung von Wörtern, andere Wörter werden neu hinzugefügt, wieder andere fallen heraus. So wurde zum Beispiel das Wort *Gebäudereinigungsfachkraft* erst vor relativ kurzer Zeit in den aktiven Wortschatz seiner Sprache aufgenommen, während das Wort *archaisch* für 'veraltet' oder 'altertümlich' kaum noch Verwendung findet. Das erste Beispiel hat aus Gründen der politischen Korrektheit nicht nur Einzug in die deutsche Sprache als neue Bezeichnung gefunden, sondern auch seinen Vorgänger ersetzt.

¹[4] beruft sich an dieser Stelle auf Weizenbaum, 1976: *Computer Power and Human Reason: From Judgement to Calculation*

Bird et. al. nahmen ebenfalls Bezug auf künstliche Sprachen. Programmiersprachen und mathematische Notationen sind, ebenso wie natürliche Sprachen, als Verbindung aus Grammatik, Wortschatz und Verwendung darstellbar. Der Unterschied zu den natürlichen Sprachen ist jedoch, dass diese Sprachen feststehen. In der Sprache der Mathematik findet außer in Fachkreisen kaum noch eine Veränderung statt. Ebenso verhält es sich mit dem täglich verwendeten Java. Mit jedem Update gibt es eine handvoll Änderungen, jedoch nimmt kaum eine dieser Änderungen Einfluss auf den alltäglichen Gebrauch der Sprache.

Die Eigenschaft von natürlichen Sprachen, dynamisch zu sein, macht es schwer, sie in kompakten Regeln klar zu formulieren. Um es dennoch zu können, wird eine Sprache in folgende sechs Teilgebiete aufgeteilt, die jedes für sich erfasst und bearbeitet werden kann: Morphologie, Syntax, Semantik, Pragmatik, Diskurs und Mehrdeutigkeit ([4]).

Für die gesprochene Sprache kommt der Punkt der Phonologie hinzu. Es folgt eine formale Definition dieser Begriffe nach den formalen Definitionen in [1].

Unter **Morphologie** verstehen wir die Lehre der Gestalten und Formen eines Objektes, in unserem Fall die Veränderung eines Wortes nach Flexion¹ und Komposition²).

Unter **Syntax** verstehen wir die Lehre vom Bau eines Satzes und dessen formaler Struktur.

Unter **Semantik** verstehen wir die Bedeutung oder den Inhalt eines Wortes, Satzes oder Textes

Unter **Pragmatik** verstehen wir die Beziehung zwischen Sprachbausteinen und ihren Anwendungen sowie das Interpretieren von Sprechen und Schreiben als Handlung.

Unter **Diskurs** verstehen wir die Gesamtheit aller sprachlichen Äußerungen eines Sprechers³

Unter **Mehrdeutigkeit** verstehen wir, dass ein Wort verschiedene Bedeutungen abhängig vom Kontext haben kann.

Unter **Phonologie** verstehen wir die Funktion der Laute in einer Sprache.

Aus Gründen der Übersichtlichkeit wird hier nur auf Morphologie eingegangen.

3 Parsing der Morphologie mithilfe von deterministischen endlichen Automaten

Zum Parsing der Morphologie sei zunächst gesagt, was unter dem Begriff der Morphologie zu verstehen ist. Die Morphologie ist die Festlegung all jener Regeln, nach denen sich ein Wort unter Flexion verändert. Dies geschieht über das Hinzufügen von *Morphemen*. Ein Morphem ist alles das, was als atomarer Bestandteil eines Wortes gelesen werden kann: Wortstamm, Flexionsendungen, Affixe. [4] führt als Beispiel das Wort *fox* sowie dessen Pluralform *foxes* an, welches sich als folgende Kette von Morphemen darstellen lässt:

¹Konjugation, Steigerung oder Deklination, je nach Wortart

²Zusammenfügen von Wörtern (Beispiel: 'Schiff' + 'Fahrt' → 'Schiffahrt')

³eine Unterhaltung

$$\begin{array}{ccc} fox & & es \\ Stamm & +N & +PL \end{array}$$

Wie man sehen kann, sind mehr Morpheme gelistet, als das Wort Flexionsbausteine hat. Zu den Morphemen zählen ebenfalls Zeichen, die grammatikalische Informationen enthalten, wie beispielsweise, dass es sich um ein Nomen handelt (+N). Mithilfe von Konventionen über die Stelligkeit, also die Reihenfolge, und Nomenklatur der Morpheme ist es möglich, wahlweise kaskadierende oder verkettete endliche, deterministische Automaten (i.F. DFAs) zu konstruieren, deren Zweck darin besteht, ein Wort in seine Morpheme zu zerlegen und uns somit mit sämtlichen grammatikalischen Informationen über dieses Wort zu versorgen.

Es liegt auf der Hand, dass eine Automatenkette, die sämtliche Wörter einer Sprache korrekt erkennen soll, sehr lang ist. Wählt man statt der Verkettung die Kaskadierung der Automaten, d.h. man konstruiert einen äquivalenten, größeren Automaten, der mehrere kleinere DFAs zusammenfasst, so wird die Kette zwar kürzer (durch genügend Rekonstruktion lässt sich jede Kette von DFAs in einem großen DFA zusammenfassen), da jedoch die Vereinigungsmenge von Knoten und Kanten um einiges Größer ist als die eines Automaten für einen einzelnen Schritt, wird das Konstrukt ungleich komplexer. Zu Veranschaulichung wird an dieser Stelle ein kleiner DFA eingefügt, der das Wort *Nachkommastellen* auf seine Morpheme reduzieren soll.

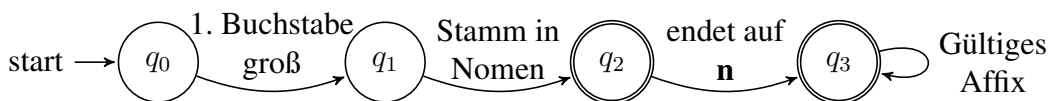


Abbildung 1: DFA zur Zerlegung des Wortes *Nachkommastellen* in seine Morpheme

Dies ist ein sehr allgemeiner DFA, um Nomen zu erkennen. Zuerst wird geprüft, ob der erste Buchstabe groß geschrieben ist, was eine Voraussetzung für Nomen in der deutschen Sprache ist und das Morphem *N* zurückliefert. Anschließend wird das Wort nach einer Zeichenkette durchsucht, die in der Liste der dem Programm bekannten Nomen steht. Da *komma* vor *stelle* steht, wird es als gültiger Stamm akzeptiert. Allerdings ist dies eine Schwachstelle dieses DFA, welcher auch fälschlicherweise *stelle* als Stamm erkennen könnte. Dieser Prozess liefert uns das Stamm-Morphem zurück.

An dieser Stelle hat der Automat erfolgreich ein Nomen erkannt und kann den Prozess beenden, wenn das Wort zu ende ist. Ist das Wort noch nicht zu ende, so prüft der Automat die verbliebenen Zeichenketten zunächst darauf, ob das Plural-Morphem vorliegt. In unserem Fall ist dieses Morphem nur der Buchstabe *n*, bei manchen Wörtern kann es sich aber auch um beispielsweise die Zeichenkette *en* handeln. Sind dann noch Zeichen übrig, werden diese darauf geprüft, ob es sich um gültige Affixe handelt. Diese werden in *q3* allesamt herausgefiltert und liefern ihr Morphem zurück. Sind nach dem Prüfen auf Affixe noch Zeichen übrig, bricht der Automat ab und akzeptiert das Wort nicht. Dies kann bei der hier gewählte Implementierung sowohl daran liegen, dass ein morphologischer Sonderfall vergessen wurde, als auch daran, dass es sich nicht um ein gültiges Wort handelt. Ist das Wort allerdings zu ende, akzeptiert der Automat das Wort

und liefert uns dessen Morphem-Kette zurück, welche in unserem Beispiel wie folgt aussieht:

N (n)ach komma stelle n
N +Pref +Stamm +Suf +PL

Wie hier zu sehen ist, ist bereits bei einem einzelnen Wort viel zu beachten. Hinzu kommt, dass hier nur eine Wortklasse abgedeckt wurde. Kommen noch andere Wortklassen, wie beispielsweise Verben, hinzu, steigt die Komplexität der Automatenkette exponentiell.

Als Ansatzpunkt wäre hier allerdings anzumerken, dass der DFA mit einer Kante von q_0 zu einem nicht angegebenen Zustand versehen werden kann, die als Beschriftung hat: " \neg 1. Buchstabe groß" (Der erste Buchstabe ist kein Großbuchstabe). Diese Kante würde eine Kaskade beginnen, durch die ein Wort durch die DFAs für sämtliche Wortklassen durchgereicht wird.

Die Operationen, die erforderlich sind, um einen String derart zu zerlegen, sind in höheren Programmiersprachen bereits enthalten. Ebenso können gewissen Morpheme mit regulären Ausdrücken (*regular expressions*/ RegEx) erkannt werden.

4 RegEx zur Vereinfachung der Parsing-Arbeit

Es ist allerdings nicht nötig, sämtliche Überprüfungen manuell zu vollziehen. Moderne Programmiersprachen implementieren ein Hilfsmittel namens *RegEx*, welches die Analyse eines gegebenen Textes erheblich vereinfacht. Die Kante, die eben als Ansatzpunkt für eine Kaskade genannt wurde (" \neg 1. Buchstabe groß") kann mithilfe einer RegEx-Überprüfung ausgelassen werden: Mithilfe des RegEx-Operators "[A-Z]" wird überprüft, ob ein Großbuchstabe vorliegt¹. Diese Überprüfung gibt einen Wahrheitswert (boolean) zurück, der dann als Argument in eine Fallunterscheidung eingesetzt werden kann:

```
1 boolean firstLetterCapital = inputWord.substring(0,1)
2                               .matches("[A-Z]");
3 if (firstLetterCapital)
4 {
5     noun = true;
6 }
7 else
8 {
9     noun = false;
10 }
```

Code 1: Überprüfung des ersten Buchstabens (lang)

Hierbei sei `inputWord` ein Wort des vom Benutzer des Programmes eingegebene Textes; die daran aufgerufene Methode `substring(0,1)` gibt den ersten Buchstaben

¹Genauer: Es wird überprüft, ob der gegebene String in der Menge aller Buchstaben von A bis Z liegt, wobei die Menge nur Großbuchstaben enthält.

dieses Wortes zurück¹; und die Methode `matches (" [A-Z] ")` ist die eben genannte RegEx-Überprüfung. Dieser Code ist vereinfachbar auf folgende einzelne Zeile:

```
1 noun = inputWord.substring(0,1).matches(" [A-Z] ");
```

Code 2: Überprüfung des ersten Buchstabens (kurz)

Der Umweg über den Zwischenspeicher `firstLetterCapital` sowie die `if`-Abfrage sind somit obsolet.

Somit steht nun fest, wie ein einzelnes Wort darauf geprüft werden kann, ob es sich um ein Nomen handelt. Um einen ganzen Satz auf Nomen zu prüfen, muss dieser in einzelne Wörter zerlegt werden. Die `String`-Klasse von Java bietet hierzu das passende Hilfsmittel:

```
1 String[] words = inputString.split("\\s+");
```

Code 3: Teilen eines Strings jedem Vorkommen von einem oder mehr Leerzeichen

Diese Code-Zeile wandelt einen gegebenen String in eine eindimensionale Matrix von Teilstrings um; der Separator zwischen den Teilstrings ist hierbei eine Kette von beliebig vielen Leerzeichen (`s+`). Durch Verkettung dieser Code-Zeile mit einer `for-each-Schleife` und der Abfrage von eben kann nun für jedes Wort einer Textzeile diese Prüfung erfolgen:

```
1 String[] words = inputString.split("\\s+");
2
3 for (String w: words)
4 {
5     noun = w.substring(0,1).matches(" [A-Z] ");
6 }
```

Hierbei ist anzumerken, dass obiger Code den Speicher `noun` bei jedem Nomen überschreibt. Obige Implementation ohne zusätzliche Speichermaßnahmen ist also wenig hilfreich. Eine Lösung ist, einen neuen Typ `word` zu erstellen, der als ein Feld das Wort selber als `String` speichert und als anderes Feld ebenjenes `boolean noun`. So kann für jedes Wort im gegebenen Eingabestring ein neues Objekt erstellt werden, welches für sich als Nomen festgelegt werden kann. Diese Lösung ist insofern erweiterbar, dass für jeden Morphemtyp (vgl. 3) ein eigenes Feld initialisiert werden kann. Ein Klassenkopf sähe beispielsweise so aus:

```
1 public class word
2 {
3     // is a name?
4     boolean noun;
5     // is a verb?
6     boolean verb;
7     // is an adjective?
```

¹Genauer: Die Zeichenkette, die am Index 0 beginnt und bis einen Index vor 1 geht (was auch 0 ist, womit die Länge der Kette genau 1 ist)

```
8   boolean adjective;  
9   // list of all affices  
10  ArrayList<String> affices = new ArrayList();  
11  
12  [...]  
13  
14 }
```

Code 4: Beispielhafter Klassenkopf für den Typ 'wort'

5 Die Arbeitsweise von ELIZA

Wie bereits erwähnt, arbeitet ELIZA mit einem *Pai matching*-Verfahren: Sie extrahiert ein Schlüsselwort aus dem ihr gegebenen Text und prüft in ihrem Speicher, ob sie eine passende Antwort kennt. Der gegebene Text wird dazu in Gruppen zerlegt. [6] führt als Beispiel den Satz *"It seems that you hate me."* ([6], S. 37) an, der in folgende Gruppen zerlegt wird:

(a) It seems that (b) you (c) hate (d) me

An dieser Stelle ist die Regelpaar-Funktion von ELIZA erwähnenswert.

Auf einen gegebenen Text, in dem die Schlüsselwörter gefunden wurden, wendet ELIZA eine *Zerlegungsregel*¹ an, um die oben gezeigte Gruppenteilung zu erhalten. Hierbei wird der gesamte Text vor dem ersten Schlüsselwort als eine Gruppe interpretiert, jedes Schlüsselwort hat eine eigene Gruppe, der Text zwischen zwei Schlüsselwörtern ist eine Gruppe und der Text nach dem letzten Schlüsselwort ist eine Gruppe.

Auf die erhaltenen Gruppen wird eine *Wiederaufbau-Regel*² angewandt. Durch einen Satz von Transformationsregeln werden Personalpronomen invertiert (*you* ↔ *me*). Anschließend werden die invertierten Pronomen und die Gruppen in anderer Reihenfolge zurück gegeben, um eine Antwort zu generieren. Dabei sind nicht alle Gruppen zwingend in der Antwort enthalten, dafür besitzt ELIZA Füllphrasen. Am Beispiel von oben:

(f) What makes you think that (d) I (c) hate (b) you

(f) ist hierbei eine von ELIZAs Füllphrasen.

Für jedes Schlüsselwort existiert ein endlicher, beliebiger Satz an Wiederaufbau-Regeln, die nacheinander angewandt werden, bis eine Regel gefunden wird, die mit der genauen Eingabe zusammen passt. Hierbei empfiehlt sich eine Prioritätenverteilung für die Regeln. In unserem Beispiel wird das Regelpaar

(0) YOU (0) ME → (0) I (3) YOU

¹Wortlaut: *Decomposition Rule* ([6], S.37)

²Wortlaut: *Reassembly Rule* ([6], S.37)

angewandt¹. Welches Regelpaar immer eine Antwort generiert und deswegen als *Fail-safe*² mit der geringsten Priorität versehen werden sollte, ist

(0) YOU (0)

Wie zu sehen ist, ist nur erforderlich, dass in der gesamten Eingabe nur das Wort 'you' enthalten ist. Also kann mit

(0) YOU (0) \longrightarrow (0) I (3)

immer genau dann eine Antwort generiert werden, wenn der Benutzer ELIZA direkt anspricht.

Bei der Implementation sollte allerdings darauf geachtet werden, ob es sinnvoll ist, dass eine Gruppe leer sein kann. Stünde bei unserem Beispiel das 'you' am Ende der Eingabe, käme von ELIZA beispielsweise "*What makes you think I.*" zurück. Dementsprechend müssen leere Gruppen entweder durch Sonderfälle in den Regeln aufgefangen werden, oder sie dürfen vom Parser gar nicht erst zugelassen werden.

6 Praktischer Nutzen der Textverarbeitung in Hinblick auf Spieleentwicklung

Anhand von ELIZA (1.2) wurde gezeigt, wie die Anfänge der Sprachverarbeitung ausgesehen haben. Mithilfe von DFAs und deren Umsetzung als Java-Code mit Einsatz von RegEx (3) wurde gezeigt, wie ein Teil der Sprachverarbeitung heute aussieht. Doch was kommt nun? Was hat man davon, dass Computer Menschen verstehen können? Ein Beispiel, welches ich an dieser Stelle aus Eigeninteresse an der Thematik vorstellen möchte, ist die Weiterentwicklung des Chat-Bots: Die Künstliche Intelligenz von Videospiel-NPCs.

Während ein Chat-Bot eine Sammlung semi-intelligenter, auf Key-Value-Pairing basierender Algorithmen zum Durchsuchen von Speicherdaten, etwa in Form einer Text-Datei, ist, ist ein NPC die Illusion einer lebenden Entität in einer fiktiven Welt. Die Abkürzung *NPC* steht für den englischen Ausdruck *Non-Player-Character*, also eine Figur, die nicht vom Spieler selber gesteuert wird.

Die Verbindung von ELIZA mit RegEx und DFAs zum Morphologie-Parsing soll einen ersten Schritt zur Konstruktion von NPCs sein, die die Möglichkeit besitzen, auf natürliche Sprache mit dynamisch generierten Antworten zu reagieren.

Die bisherige Interaktion mit NPCs beläuft sich darauf, dass ein Spieler keine oder kaum Möglichkeiten hat, sich an einem Gespräch zu beteiligen. Entweder ihm wird eine Antwort in den Mund gelegt, er bekommt eine kleine Auswahl von meist nicht mehr als sechs Antwortmöglichkeiten, oder sein Charakter nimmt gar nicht am Gespräch teil.

¹Notation nach [6]; (0) entspricht einem 'Don't Care', mit (3) ist die dritte Gruppe der Eingabe gemeint

²Sie wird ausgeführt, wenn alles andere scheitert

Darf sich der Spieler beteiligen, hat er meist sehr begrenzte Möglichkeiten. Drei häufig verwendete Mechaniken können herausgearbeitet werden:

Parsing. Diese Methode wird bisher bei den meisten Text-Adventures¹ verwendet. Der Vorteil ist, dass sie mit den in dieser Arbeit beschriebenen einfachen Mitteln leicht umgesetzt werden kann und aufgrund ihres Alters am meisten ausgereift ist. Der Nachteil ist, dass bei schlechter Implementierung nur eine spezifische Kombination von Schlüsselwörtern überhaupt zu einem Ergebnis führt (für ein Beispiel vergleiche Anhang 1 Abbildung 2).

Optionsdialog.² Dem Spieler werden eine handvoll Antworten vorgegeben. Diese Methode kann unter Anderem in den *Elder Scrolls*-Spielen betrachtet werden (Beispiel siehe Anhang 1 Abbildung 3).

Vorteilhaft bei dieser Methode ist, dass die Möglichkeiten klar sind und der Spieler nicht experimentieren muss.

Der Nachteil ist, dass der Gesprächsverlauf von vornherein fest steht und es viel Arbeit ist, viele Gesprächsoptionen zu implementieren, wobei der Aufwand exponentiell zu der Anzahl der Akteure, mit denen der Spieler redet, wächst.

Das Hybrid-System. Während [3] an dieser Stelle zwei Subsysteme nennt, ist nur eines von Interesse. Das zweite System, welches hier genannt wird, ist der tatsächliche Hybrid, der eine handvoll Schlüsselwörter vorgibt, dem Spieler allerdings auch die Möglichkeit zur Eingabe einer Schlüsselwörter lässt³.

Der Vorteil ist die große Freiheit neben den Dialog-Optionen, was dem Spieler erlaubt, im Gespräch zu experimentieren und trotzdem schnell alle gewünschten Informationen zu erhalten.

Der Nachteil bei schlechter Implementierung ist, dass ein Spieler möglicherweise ein Schlüsselwort errät, welches er zu diesem spezifischen Zeitpunkt nicht kennen sollte. Über Flags lässt sich dies allerdings abfangen.

Wie gerade gezeigt, findet die Textverarbeitung Anwendung in einem breiten Feld von Spielen. Was in den meisten Spielen jedoch fehlt, ist eine Implementierung von ELIZA, weswegen die meisten Gespräche und Handlungen starr und vorgeschrieben sind. Die Verbindung der aktuell verwendeten Textverarbeitung mit ELIZA ermöglicht ein dynamischeres Spielgeschehen und eine größere Immersion in die Spielwelt.

7 Schlussfolgerung

Mithilfe der hier erörterten Arbeitsweise ist ein erster Schritt getan, um den Spieler natürliche Sprache anwenden zu lassen, um mit NPCs zu kommunizieren:

Die mithilfe von RegEx und der Java-String-Klasse umgesetzten DFAs führen alle Wörter auf ihre Stammformen und Morpheme zurück, woraufhin ein auf ELIZA-basierendes

¹Der Spieler bekommt Informationen über Text und reagiert mit Text

²[3] nennt diese Methode "Menu-Driven Conversations"

³[3] nennt ebenfalls ein System, welches er *All-Keyword-System* nennt. Der Spieler kann mit jedem Akteur über alles reden. Erhält er neue Informationen, aktualisiert sich die Liste der Gesprächsoptionen. Der Nachteil ist die starre Sammlung an Gesprächsoptionen. Außerdem geht Realismus verloren, wenn man einen Akteur auf ein Gerücht ansprechen kann, von dem er rein logisch nicht wissen kann.

Programm Antworten generieren kann.

Natürlich ist damit nur ein minimaler Teil der natürlichen Sprache abgedeckt. Die Antworten müssen dem Programm immernoch schablonenhaft vorgegeben werden. Ohne Implementation der Analyse der restlichen in 2 genannten Aspekte natürlicher Sprache hat das Programm keine Möglichkeit, zu wissen, wovon der Spieler tatsächlich redet. Allerdings kann mithilfe der Schablonentechnik von ELIZA (5) bereits ein größtenteils von der Eingabe des Spielers abhängiges Gespräch stattfinden. So kann ein NPC sympathisch auf Aussagen des Spielers reagieren; oder den Kontrollfluss im Handlungsablauf eines Spiels anpassen, wenn ein bestimmtes Wort gesagt wird – wird kein Schlüsselwort gefunden, wird ein Basis-Handlungsstrang verfolgt.

Eine Implementierung dieses Verfahrens erlaubt eine Immersion in Spiele, wie sie bisher nur wenig gesehen wurde. Ein Spieler hat mehr Freiheit als je zuvor, und die Welt passt sich ihm dynamisch an.

Die aufgrund des aktuellen Technologiestandes naheliegende Erweiterung ist zunächst die Phonologie, um aus gesprochener Sprache einen Text zu erhalten, der mit eben diesem Verfahren analysiert werden kann¹. Erst danach lohnt es sich, die vollständige Implementierung des Sprachverständnisses anzustreben.

¹Ein großer Schritt in diese Richtung ist die *XBOX Kinect*-Technologie, mit der ein Spieler zum Beispiel im Spiel *Skyrim* seinen Begleitern simple Befehle wie "Greife an" oder "Öffne Tür" geben kann - über Sprachsteuerung. Im Gegensatz dazu ist Microsoft mit *Cortana* einen Schritt zurück gegangen, da sich die Benutzung dieses Programmes als langwierig und anstrengend gestaltet ([5], Abschnitt 6: Cortana)

Literatur

- [1] *Duden - Die deutsche Rechtschreibung*, volume 1 of *Duden 1-12*. Dudenredaktion, 26 edition, October 2014.
- [2] Ewan Klein; Edward Loper; Steven Bird. *Natural language processing with Python*. O'Reilly Media, Inc, 2009.
- [3] Unknown Author (edited by Amit Patel). Npc conversations. Stanford University, Department of Computer Science (<http://www-cs-students.stanford.edu/~amitp/Articles/NPC-Conversation.html>).
- [4] James H. Martin; Daniel Jurafsky. *Speech and Language Processing*. Prentice Hall, 2009.
- [5] Hauke Stieler. Windows 10: Warum microsoft sich verzaehlt hat (<http://hauke-stieler.de/blog/2015/06/windows-10-warum-microsoft-sich-verzaehlt-hat/>). Blog-post, June 2015.
- [6] Joseph Weizenbaum. Eliza - a computer program for the study of natural language communication between man and machine. *Communications of the ACM*, 9(1):36–45, January 1966.

Anhang 1: Grafiken

TAKE THE WINE

"I don't know what a WINE is"

TAKE THE FLASK

"I don't know what a FLASK is"

TAKE THE DAMN BOTTLE

"I don't know what a DAMN is"

TAKE THE BOTTLE

"You have taken the bottle."

Abbildung 2: Beispieldialog bei der Text-Parsing-Methode; Quelle: Screenshot eines Auszuges aus [3]

Angry Orc: "Hey! Nobody gets in here without the password."

A. "Yes. The password. I know it."

B. "The password? Oh, I forgot it. It's downstairs. I'll be right back."

C. "Password? I don't need no stinkin' password!"

Abbildung 3: Beispiel für Antwortmöglichkeiten im Optionsdialog; Quelle: Screenshot eines Auszuges aus [3]