Bauhaus-Universität Weimar
Fakultät Medien
Studiengang Mediensysteme

Einsatz heuristischer Suchverfahren zur Erzeugung eines Akrostichons

Masterarbeit

Christof Bräutigam
Geboren am 22.05.1979 in Rudolstadt

Matrikelnummer 40008

1. Gutachter: Prof. Dr. Benno Stein
Betreuer: Dr. Matthias Hagen

Datum der Abgabe: 5. September 2012

Erklärung

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Weimar, den 5. September 2012

.....
Christof Bräutigam

Kurzfassung

Diese Arbeit behandelt die Konstruktion eines Akrostichons. Ein Akrostichon liegt vor, wenn die Anfangsbuchstaben der Zeilen eines Textes, gelesen von oben nach unten, eine sinnvolle Botschaft bilden. Akrosticha treten in natürlichen Texten, wie etwa dieser Zusammenfassung, nur extrem selten auf. Wir entwickeln ein Verfahren um ein Akrostichon in einem natürlichen Text automatisch zu erzeugen. Dabei soll sowohl der Text als auch das Akrostichon beliebig gewählt werden können.

Wir betrachten diese Aufgabe als Suchproblem. Unsere Idee ist, eine Menge von Variationen des Textes zu erzeugen und nach einer zu durchsuchen, die das Akrostichon enthält. Dazu entwickeln wir Operatoren, das heißt Methoden, um Text systematisch zu verändern. Aus der kombinierten Anwendung der Operatoren auf den Text entsteht der Suchraum, den wir mit einer Best-First-Suche durchsuchen. Die Herausforderung besteht darin, trotz begrenzter Ressourcen mit einer großen Menge von Objekten im Suchraum zu arbeiten und die Suche intelligent zu steuern, da sonst eine Lösung nicht im Bereich des technisch Machbaren liegt. Wir implementieren ein System um diese Idee zu demonstrieren. In Experimenten zeigt sich, dass dieses System beliebige Akrosticha in beliebigen Texten in etwa 14% aller Fälle erzeugen kann. Für spezielle Akrosticha liegt die Erfolgsquote deutlich höher.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Gegenstand dieser Arbeit ist das Erstellen eines Akrostichons in einem Text. Ein Akrostichon ist ein literarisches Stilmittel, bei dem die Anfangsbuchstaben der Zeilen eines Textes von oben nach unten gelesen ebenfalls einen sinnvollen Text ergeben, beispielsweise einen Namen oder eine kurze Botschaft. Für ein Akrostichon ist also nicht nur der Inhalt, sondern auch die Struktur des geschriebenen Textes von Interesse. Ein Beispiel ist das folgende Gedicht:[1]

```
Himmel färbt sich grau
Eimerweise Regen
Raue Winde stürmen
Blätter tanzen wild
Sommer ist vorüber
Traurig nun mein Herz
```

Die Zeilenanfänge des Gedichtes bilden das Wort „Herbst“ als Akrostichon. Akrosticha sind häufig auf diese Weise in Gedichten eingebaut. Bekannte Gedichte mit Akrosticha sind unter anderem von Edgar Allen Poe und Lewis Carroll überliefert[2].

1.1 Varianten von Akrosticha

Akrosticha werden in mehreren Varianten verwendet. So können auch die ersten Worte jeder Strophe eines Gedichtes oder Liedes einen sinnvollen Text

[1] Quelle: `http://de.wiktionary.org/wiki/Akrostichon`, letzter Abruf 22.8.2012.

[2] Der englische Wikipedia-Artikel enthält einige Beispiele: `http://en.wikipedia.org/wiki/Acrostic`, letzter Abruf 22.8.2012.

> For some time now I have lamented the fact that major issues are overlooked while many unnecessary bills come to me for consideration. Water reform, prison reform, and health care are major issues my Administration has brought to the table, but the Legislature just kicks the can down the alley.
>
> Yet another legislative year has come and gone without the major reforms Californians overwhelmingly deserve. In light of this, and after careful consideration, I believe it is unnecessary to sign this measure at this time.

Abbildung 1.1: Ausschnitt des Briefs von Gouverneur Schwarzenegger an den Abgeordneten Ammiano. Die beiden Absätze enthalten ein Akrostichon.

bilden, ein Beispiel hierfür ist „Het Wilhelmus“, die Nationalhymne der Niederlande,[3] oder die Anfangsbuchstaben von Textabsätzen ergeben eine Botschaft.[4] Eine weitere, sehr verbreitete Form ist die Verwendung als Merkspruch, beispielsweise bezeichnen die Anfangsbuchstaben von „Ein Anfänger Der Gitarre Hat Eifer“ die Töne der Gitarrensaiten. Noch eine andere Möglichkeit, Akrosticha zu bilden, sind die Anfangsbuchstaben aufeinander folgender Sätze. Vielleicht eines der schönsten Beispiele findet sich in Douglas R. Hofstadter's Buch „Gödel, Escher, Bach“, in dem die Anfangsbuchstaben eines mehrseitigen Dialogs ein selbstreferentielles Akrostichon ergeben, dessen Worte in ihren Anfangsbuchstaben wiederum ein Akrostichon enthalten.

1.2 Natürliche Akrosticha

Alle bisher genannten Beispiele haben eine Gemeinsamkeit: Sie wurden absichtlich mit dem Text konstruiert, das heißt, schon bei der Erzeugung des Textes hatten die Autoren alle Freiheiten, die Worte entsprechend zu wählen. Dass ein Akrostichon zufällig in einem „natürlichen“ Text auftaucht, ist sehr unwahrscheinlich.

Im Rahmen dieser Arbeit wollen wir Akrosticha in „natürlichen“ Texten erzeugen, etwa in einem Zeitungsartikel, einer Email oder einem Blogeintrag. Wir betrachten dabei speziell die Form eines Akrostichons, bei der die Anfangsbuchstaben der Textzeilen von oben nach unten gelesen eine sinnvolle Botschaft ergeben. Der erste Buchstabe des Textes soll zudem auch der erste Buchstabe des Akrostichons sein.

Ein Beispiel für diese Art von Akrostichon findet sich in einem Brief des

[3] Quelle: `http://de.wikipedia.org/wiki/Het_Wilhelmus`, letzter Abruf 22.8.2012.

[4] Ein Beispiel für diese Form eines Akrostichons findet sich in einem Memo des CEO von Sun, Jonathan Schwartz, dessen erste sieben Absätze mit den Buchstaben „Beat IBM“ beginnen. Quelle: `http://allthingsd.com/20100121/sun-ceo-go-oracle-internal-memo/`, letzter Abruf 22.8.2012.

(damaligen) kalifornischen Gouverneurs Arnold Schwarzenegger an den Abgeordneten Tom Ammiano. Ein Ausschnitt des Briefs ist in Abbildung 1.1 dargestellt. Dieser Brief, versendet im Oktober 2009, enthält außer der kurzen Begründung für die Ablehnung der Finanzierung eines Bauprojektes auch ein Akrostichon. Das Akrostichon ist eine unfreundliche Botschaft an den Adressaten, laut offizieller Aussage handelt es sich dabei aber gar nicht um eine Botschaft, sondern um einen „unglücklichen Zufall“.[5] Dies wäre der einzige uns bekannte Fall, in dem ein Akrostichon dieser Form „zufällig“ in einem Text entstanden ist.

1.3 Automatische Erzeugung

Wir wollen uns damit beschäftigen, wie ein solches Akrostichon automatisch erzeugt werden kann, wenn sowohl das Akrostichon, als auch der Text in dem es auftauchen soll, vorgegeben werden. Wir müssen den vorgegebenen Text daher so verändern, dass das Akrostichon darin erzeugt wird. Die Änderungen sollen den Sinn des Textes möglichst wenig beeinflussen. Wir haben dieses Problem als Suchproblem formuliert. Unsere Idee ist, den Text systematisch zu modifizieren, beispielsweise durch Einfügen oder Ersetzen von Wörtern, und somit eine Menge von Textvarianten zu erzeugen. Diese Menge interpretieren wir als Suchraum, der mit bekannten Suchverfahren nach einem Ziel – einer Textvariante mit dem gewünschten Akrostichon – durchsucht werden kann.

Zur Demonstration der Idee wird ein Suchsystem entwickelt und implementiert, das in der Lage ist, Akrosticha zu erzeugen. Das System wird so entworfen, dass neue Ideen und Methoden einfach integrierbar sind.

1.4 Verwandte Themengebiete

Thematisch bewegt sich diese Arbeit in der Schnittmenge verschiedener Forschungsgebiete der Informatik. Mit der Formulierung als Suchproblem siedeln wir die Erzeugung eines Akrostichons im Gebiet der *Künstlichen Intelligenz* (KI) an. Hier nutzen wir vor allem die Erkenntnisse im Bereich der heuristischen Suchverfahren.

Methoden zur automatischen Verarbeitung von natürlichsprachlichem Text werden in den Bereichen *Computerlinguistik* (CL) beziehungsweise *Natural Language Processing* (NLP) erforscht. Die Aufgabe, Texte sinnerhaltend umzuformulieren, so dass gleiche Sachverhalte mit anderen Worten ausgedrückt

[5] `http://www.huffingtonpost.com/2009/10/30/schwarzenegger-f-bomb-in_n_340579.html`, letzter Abruf 22.8.2012.

werden können ist eine Problemstellung der CL, die allgemein als *Paraphrasierung* bezeichnet wird. Diese Aufgabe scheint sehr eng verwandt mit unserer Problemstellung, daher diskutieren wir die Erkenntnisse der Paraphrasing-Forschung im Kontext unserer Betrachtung zur Textveränderung. Die bekannten Verfahren sind für unsere speziellen Anforderungen leider unpraktisch, daher entwickeln wir eigene Methoden.

Die Schnittmenge der Forschungsgebiete KI und CL ist bislang klein, birgt aber großes Potenzial, beispielsweise in der Erforschung und Entwicklung automatisierter Dialogsysteme oder in der maschinellen Übersetzung. Uns ist bisher keine andere Arbeit bekannt, die thematisch ähnlich gelagert ist, wir wollen mit dieser Arbeit also auch einen ersten Schritt in diesem Bereich machen.

1.5 Gliederung

Der Hauptteil der Arbeit ist wie folgt gegliedert. Kapitel 2 führt in die Begriffe und Konzepte der heuristischen Suche in Zustandsräumen ein. Die dort vorgestellten Suchverfahren bilden die Grundlage unserer Methode zur Erzeugung von Akrosticha. Da wir davon ausgehen, dass wir den gegebenen Text verändern müssen, um das gewünschte Akrostichon zu erzeugen, müssen wir Methoden zur Textveränderung entwickeln, die im Rahmen eines Suchverfahrens als Operatoren verwendet werden können. Die Operatoren werden in Kapitel 3 diskutiert. Kapitel 4 baut auf den Begriffen und Ideen der beiden vorhergehenden Kapitel auf. Wir beschreiben, wie wir das Problem des Erzeugens eines Akrostichons als Suchproblem umsetzen und wie das Suchsystem aufgebaut ist. Mit dem Suchsystem führen wir Experimente durch, um die Idee zu demonstrieren und das System zu evaluieren. Der Aufbau und die Ergebnisse der Experimente werden in Kapitel 6 beschrieben. Kapitel 7 fasst die Arbeit zusammen und gibt einen Ausblick auf mögliche zukünftige Entwicklungen.

Kapitel 2

Suche

Der Einsatz von Suchverfahren zur Problemlösung ist eine grundlegende Technik in der Künstlichen Intelligenz (KI). Dieses Kapitel führt in die Konzepte und Begriffe der Suche ein. Wir orientieren uns dabei an *Heuristics: Intelligent Search Strategies for Computer Problem Solving* von Judea Pearl [Pea84, Kap. 1] sowie *Artificial Intelligence: A Modern Approach* von Stuart Russel und Peter Norvig [RN10, Kap. 3].

2.1 Formulierung von Suchproblemen

Unsere Problemstellung ist die Erzeugung eines Akrostichons in einem Text. Wir wollen diese Aufgabe als Suchproblem formulieren. Jede Problemlösungsaufgabe lässt sich allgemein als Suche betrachten [Pea84]: „Finde ein Objekt mit bestimmten Eigenschaften in einem Suchraum.“ Dabei lassen sich verschiedene Problemtypen unterscheiden:

- *Bedingungserfüllungsproblem* : Ein Lösungskandidat muss Bedingungen erfüllen und soll mit minimalem Aufwand gefunden werden.
- *Optimierungsproblem* : Ein Lösungskandidat muss Bedingungen erfüllen und hat im Vergleich zu anderen Kandidaten noch besondere Qualitäten. Der Suchaufwand soll ebenfalls minimal sein.

In der Literatur werden grundlegende Konzepte zur Formulierung von Suchproblemen mit den folgenden Begriffen bezeichnet:

- *Datenbasis* : Codierung von Objekten im Suchraum S.
- *Operator* : Mechanismus, um ein Objekt aus S in ein anderes zu transformieren.

- *Steuerungsstrategie* : Effektives Verfahren, um die Reihenfolge möglicher Transformationen festzulegen, mit dem Ziel, das gesuchte Objekt zu finden.

Die Suche ist *systematisch* [Pea84], wenn die Steuerungsstrategie

1. alle Objekte in S betrachten kann (Vollständigkeit)
2. jedes Objekt aus S nur einmal betrachtet (Effizienz).

Der Suchraum S wird durch die Operatoren aufgespannt. Die Datenbasis muss Codierungen (das heißt Repräsentationen) für Lösungskandidaten enthalten, denn wenn Lösungen nicht repräsentierbar sind, können sie auch nicht gefunden werden. Wünschenswert ist aber auch die Möglichkeit der Codierung von Teillösungen. Eine Teillösung lässt sich auf zwei Arten betrachten. Zum einen als Baustein zur schrittweisen Konstruktion eines kompletten Lösungskandidaten. Zum anderen als Repräsentation einer Teilmenge $S' \subset S$ von Lösungskandidaten, nämlich aller Lösungen, die die Teillösung enthalten. Mit dieser zweiten Sichtweise kann man den Problemlösungsprozess als Folge von Transformationen einer Teilmenge $S' \subset S$ in eine Teilmenge $S'' \subset S'$ betrachten. Verdeutlicht wird dies in einem Beispiel in Abschnitt 2.3.2. Die Codierung einer Teilmenge S' eines Suchraumes muss dabei eindeutig sein. Die konstruktive Sichtweise wird bei der Beschreibung von Suchverfahren eingenommen, während die Betrachtung von Teilmengen bei der Rechtfertigung von Suchverfahren hilfreich ist.

Die Anforderungen an eine systematische Suche lassen sich auf die Betrachtung von Teilmengen $S' \subset S$ übertragen [Pea84]:

1. Alle Lösungskandidaten aus S müssen in der Vereinigungsmenge aller Teilmengen S' enthalten sein. (Vollständigkeit)
2. Falls eine Teilmenge $S' \subset S$ bei der Lösungssuche ausgeschlossen wurde, darf sich durch Transformation der übrigen Teilmengen kein Lösungskandidat aus S' erzeugen lassen. (Effizienz)

Unter diesen Anforderung ist die Verfeinerung die einzige zulässige Transformation, d.h. jede Transformation einer Teilmenge S' erzeugt eine echte Teilmenge $S'' \subset S'$.

Eine Teillösung repräsentiert nicht nur einen Teil einer Lösung, sondern auch ein Teil- oder Restproblem, also einen noch ungelösten Teil des Problems. Dieses Teilproblem ist implizit durch die Teillösung codiert, es ist aber oft von Vorteil, für das Teilproblem eine eigene Codierung einzuführen. Die explizite Codierung eines Teilproblems wird als *Zustand* bezeichnet.

Die Menge aller Teilprobleme, die sich durch die Anwendung der Operatoren auf die Datenbasis erzeugen lassen, bildet den *Zustandsraum*. Verknüpft man die Objekte des Zustandsraumes mit gerichteten Kanten, die mit den angewandten Operatoren markiert sind, so erhält man einen *Zustandsraumgraph*. Eine Lösung wird im Zustandsraumgraph durch einen Pfad von einem Startzustand s zu einem Zielzustand repräsentiert.

Zu beachten ist, dass die Codierung einer Teillösung, also eines Pfades von s zu einem bestimmten Zustand s', ausreichend zur Beschreibung der Gesamtsituation ist, die Zustandsinformation für s' hingegen nicht, denn ein Zustand kann auf verschiedenen Pfaden erreicht werden.

Suchverfahren lassen sich hinsichtlich der Information, die zur Steuerung des Suchprozesses herangezogen wird, unterscheiden. Wenn die Steuerungsstrategie nur die in der Problemspezifikation gegebene Information bei der Auswahl der Operatoren benutzt, bezeichnet man die Suche als *uninformiert*. Oft lässt sich aus der Problembeschreibung keine Information zur Operatorauswahl entnehmen, beispielsweise bei den in Abschnitt 2.2 genannten Suchproblemen. Wenn der Steuerungsstrategie aber Information zur Verfügung steht, bezeichnet man die Suche als *informiert*. Woher die Information kommt, ist unerheblich, es kann beispielsweise gelerntes Wissen aus vorherigen Suchvorgängen oder spezielles Domänenwissen sein. Eine informierte Suchstrategie wird auch *Heuristik* genannt.

Eine Heuristik bewertet einen Zustand und schätzt die Schwierigkeit des Restproblems oder die Erfolgsaussichten in diesem Zustand ab. Dies ist ein Grund, warum die explizite Codierung des Zustandes sinnvoll ist, obwohl die Zustandsinformation immer auch aus der Codierung einer Teillösung hergeleitet werden kann. Die Herleitung jedoch kann sehr aufwändig sein und die Codierung einer Teillösung inklusive Zustand soll dann die Berechnung von Heuristiken vereinfachen.

2.2 Beispiele für Suchprobleme

Die im vorigen Abschnitt eingeführten Begriffe und Konzepte werden anhand einiger in der KI-Literatur verbreiteter Beispielprobleme erläutert. Zunächst folgt eine informelle Problembeschreibung, im Anschluss betrachten wir die Formulierung als Suchproblem.

Das 8-Damen-Problem Auf einem Schachbrett sollen 8 Damen so angeordnet werden, dass keine Dame eine andere schlagen[1] kann.

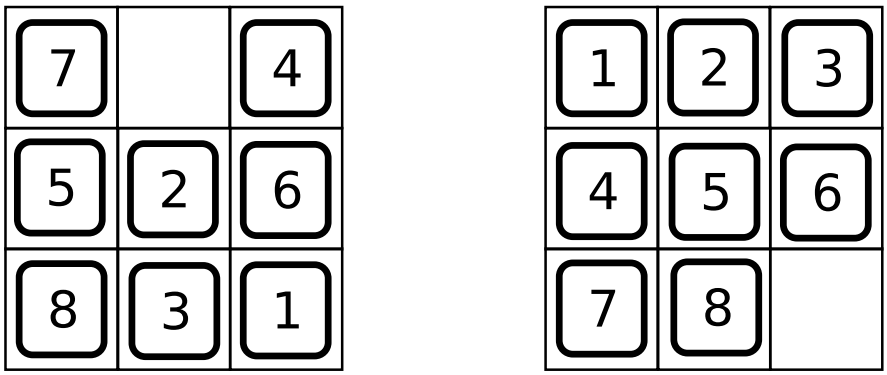
[1] Eine Dame kann beim Schach horizontal, vertikal und diagonal in beliebiger Entfernung schlagen, wenn der Weg nicht von einer anderen Figur blockiert ist.

Das 8-Damen-Problem gehört zu den am besten untersuchten Problemen[2] der Mathematik und Informatik. Die Problemstellung sagt nichts über den Startzustand aus. Bei der Formulierung als Suchproblem steht es uns also frei mit einem leeren Brett zu beginnen oder schon einige Damen auf dem Brett verteilt zu haben. Die Lösung wird nicht explizit benannt, stattdessen wird eine Bedingung definiert, die erfüllt sein muss, um einen Zustand als Lösung zu akzeptieren. Das 8-Damen-Problem ist also ein Bedingungserfüllungsproblem. Es ist bekannt, dass 92 Stellungen existieren, die die Bedingung erfüllen. Die Bedingung kann außer zum Erkennen eines Zielzustandes auch bei der Formulierung der Operatoren genutzt werden. Beispiele dafür werden in Abschnitt 2.3.1 gezeigt.

Das 8-Puzzle Dieses Problem gehört zur Klasse der Schiebepuzzle: In einem quadratischen Rahmen mit Platz für 3×3 Teile sind 8 bewegliche Teile mit den Bezeichnungen 1 bis 8 eingebracht, eine Stelle ist frei. Ein Teil, das horizontal oder vertikal neben der freien Stelle liegt, darf auf diese verschoben werden. Im Zielzustand sind die Zahlen beginnend in der oberen Reihe und von links nach rechts aufsteigend angeordnet. Die Aufgabe lautet, einen zufälligen, lösbaren Startzustand durch Verschieben der Teile in den Zielzustand zu überführen. Lösbar heißt für ein Schiebepuzzle, dass der Zielzustand auch durch eine Reihe von Verschiebungen aus dem Startzustand erreichbar ist. Dies ist bei einem Schiebepuzzle nicht für alle Zustände der Fall. Insbesondere war eine Ende des 19. Jahrhunderts populäre Version, das so genannte 15-Puzzle unlösbar, denn die Position der 14 und der 15 waren vertauscht, und es existiert keine Zugfolge, die diese Positionen korrigiert [Arc99]. Für beliebige n-Puzzle mit gleicher Spielmechanik läasst sich zeigen, dass nur die Hälfte aller möglichen Zustände lösbar ist. Das 8-Puzzle hat $\frac{9!}{2} = 181.440$ lösbare Zustände. Die Abbildung 2.1 zeigt einen möglichen Startzustand und den Zielzustand.

Generelle n-Puzzle sind beliebte Testprobleme für Suchverfahren. In der üblichen Problemstellung ist nur der Zielzustand zu erreichen, es handelt sich also auch um ein Bedingungserfüllungsproblem. Eine Lösung besteht darin, eine Sequenz von Zügen zu finden, die vom Start zum Ziel führen. Probleme dieser Art bezeichnet man auch als *Pfadsuche*. Das Problem kann aber auch als Optimierungsproblem formuliert werden: „Finde die Lösung mit der geringsten Anzahl Bewegungen.“ Für diese Formulierung wurde gezeigt, dass das Problem NP-hart ist [RW86], es existieren also keine effizienteren Lösungsverfahren als die gezeigten Suchverfahren.

[2] Die Website `http://www.liacs.nl/~kosters/nqueens/` führt eine ständig aktualisierte Liste von Veröffentlichungen mit Bezug zum n-Damen-Problem.



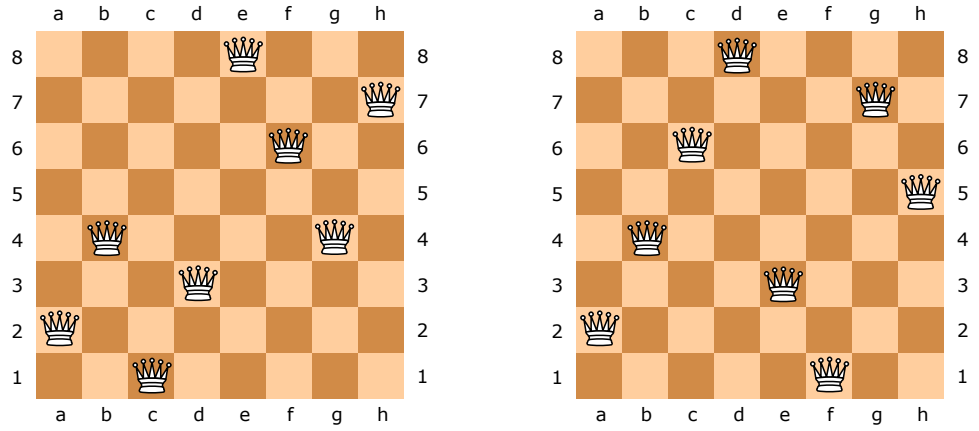
(a) typischer Startzustand (b) Zielzustand

Abbildung 2.1: Ein typischer Startzustand des 8-Puzzle 2.1(a) und der Zielzustand 2.1(b).

2.3 Problemrepräsentation

Wir betrachten, wie die genannten Probleme formell beschrieben werden können. Dazu werden die in Abschnitt 2.1 beschriebenen Konzepte Datenbasis und Operator weiter differenziert. Eine Problembeschreibung besteht aus folgenden Komponenten [RN10]:

- *Startzustand* : Repräsentiert die Ausgangssituation.
- *Operatoren* : Die zur Verfügung stehenden Operatoren und eine Funktion $operators(s)$, die, gegeben einen bestimmten Zustand s, die Menge der in diesem Zustand anwendbaren Operatoren liefert.
- *Transitionsmodell* : Eine Beschreibung der Zustandstransformation durch eine Funktion $result(s, o)$, die, gegeben einen bestimmten Zustand s und einen Operator o, den Folgezustand s' liefert.
- *Zieltest* : Eine Beschreibung einer Lösung in Form eines Prädikats $goal(s)$, das einen Zustand s als Lösung akzeptiert oder nicht.
- *Pfadkosten* : Eine Beschreibung, wie der Aufwand für eine bestimmte Teillösung zu ermitteln ist, dargestellt als Funktion $g(s)$. Für die Suche in einem Zustandsraum gehen wir davon aus, dass sich g als Summe nicht negativer Schrittkosten $cost(s, s')$ für beliebige Zustände s und Nachfolgezustände s' berechnen lässt, so dass gilt $g(s') = g(s) + cost(s, s')$.



(a) (a2, b4, c1, d3, e8, f6, g4, h7)

(b) (a2, b4, c6, d8, e3, f1, g7, h5)

Abbildung 2.2: Zwei mögliche Zustände für das 8-Damen-Problem. 2.2(a) ist ein Kandidat, aber keine Lösung, da sich die Damen auf d3 und h7 gegenseitig schlagen können. 2.2(b) erfüllt die Zielbedingung und ist eine Lösung.

Eine Anmerkung zu den Operatoren: In den folgenden Beispielen wird teilweise ein Operator formuliert, der verschiedene Zustände erzeugen kann. Dies kann äquivalent auch als Menge von Operatoren betrachtet werden, die jeweils genau eine Zustandsmodifikation bewirken. Zur Formulierung der Operatoren gehören immer auch die Bedingungen, unter denen sie angewendet werden dürfen, denn üblicherweise kann nicht jeder Operator auf jeden Zustand angewendet werden. Im 8-Puzzle beispielsweise ist für jeden Zustand die Anzahl der verfügbaren Operatoren dadurch beschränkt, welche Teile sich in welcher Position relativ zur freien Stelle befinden. Die Funktion $operators(s)$ liefert für einen bestimmten Zustand s diese anwendbaren Operatoren.

2.3.1 8-Damen-Problem

Ein Zustand im 8-Damen-Problem sind beispielsweise 8 Damen, die beliebig auf die 64 Felder des Schachbretts verteilt sind. Eine naheliegende Codierung ist ein 8-Tupel, beispielsweise (a2, b4, c1, d3, e8, f6, g4, h7).[3] Eine mögliche Lösung lautet (a2, b4, c6, d8, e3, f1, g7, h5). Die beiden Zustände sind in Abbildung 2.2 dargestellt.

[3]Die Elemente folgen der im Schach üblichen Notation. Die Spalten (genannt „Linien“) werden mit den Buchstaben a bis h und die Reihen mit den Ziffern 1 bis 8 bezeichnet. Auf die Notation der Figur können wir verzichten, da in diesem speziellen Problem nur Positionen für Damen notiert werden.

Das Problem lässt sich beispielsweise wie folgt formell beschreiben, wobei diese erste Beschreibung schlecht gewählt ist, wie wir im Weiteren sehen werden:

- Startzustand: 8 zufällig auf dem Schachbrett verteilte Damen.
- Operatoren: Verteile alle Damen zufällig neu auf dem Brett.
- Zieltest: Keine Dame kann eine andere schlagen.
- Pfadkosten: Die Kosten sind irrelevant.

Ein Suchverfahren, nennen wir es „Zufallssuche“, das mit dieser Problembeschreibung arbeitet, lässt sich so formulieren:

1. Erstelle einen Lösungskandidaten (Anwendung eines Operators).
2. Teste ob eine Lösung gefunden wurde (Anwendung des Zieltests). Wenn keine Lösung gefunden wurde, gehe zu Schritt 1.

Das Verfahren verletzt eine Anforderungen der systematischen Suche (vgl. Abschnitt 2.1), da im Suchprozess auch schon bekannte Stellungen erzeugt werden, und ist daher ineffizient. Die Problembeschreibung, und dabei speziell die Formulierung der Operatoren, lässt aber auch keine bessere Methode zu.

Betrachten wir den Suchraum, der hier aufgespannt wird. Die Operatoren können alle Kombination von 8 Damen auf 64 Feldern erzeugen, die Suchraumgröße berechnet sich als $\binom{64}{8} = 4.426.165.368$.

Eine kurze Aufwandsabschätzung: Betrachtet man Zufallssuche als Laplace-Experiment,[4] lässt sich berechnen, wie viele Zieltests man im Durchschnitt durchführen muss, um mit einer bestimmten Wahrscheinlichkeit p eine Lösung zu finden. Für $p > 0{,}5$ etwa braucht man mehr als $24{,}05 \times 10^6$ Tests. Dabei wird angenommen dass die Lösungen gleichmäßig über den Suchraum verteilt sind.

Obwohl die Zufallssuche ein ineffizientes Verfahren ist, demonstriert sie ein Prinzip der Problemlösung, das als „generate-and-test“ bezeichnet wird: Lösungskandidaten werden erzeugt und auf ihre Eignung als Lösung überprüft.

Wir verbessern die Problembeschreibung, indem wir mehr Information aus der Problemstruktur einfließen lassen. Aus den Bedingungen an eine Lösung lässt sich beispielsweise entnehmen, dass keine Lösungen existieren, bei denen mehr als eine Dame in einer Reihe steht. Wir passen den Startzustand an und formulieren die Operatoren so, dass keine Stellung mit mehr als einer Dame

[4] Ein Zufallsexperiment über einer Ergebnismenge mit diskreter Gleichverteilung wie beispielsweise das Werfen eines fairen Würfels.

pro Reihe erzeugt wird. Dies erreichen wir, indem wir die Damen nur „entlang der Spalten“ bewegen:

- Startzustand: 8 Damen, eine in jeder Reihe.
- Operatoren: Verschiebe die Dame in Reihe 1 um ein Feld nach rechts. Wenn eine Dame über den Rand verschoben wird, setzte die entsprechende Dame wieder in Spalte a und verschiebe zusätzlich die Dame in der Reihe darüber um ein Feld nach rechts.
- Zieltest: Keine Dame kann eine andere schlagen.
- Pfadkosten: Die Kosten sind irrelevant.

Wir nutzen auch ein neues Suchverfahren, „Lineare Suche“ basierend auf dieser Problembeschreibung, das prinzipiell genauso vorgeht wie Zufallssuche:

1. Erstelle einen Lösungskandidaten (Anwendung eines Operators).
2. Teste ob eine Lösung gefunden wurde (Anwendung des Zieltests). Wenn nicht, gehe zu Schritt 1.

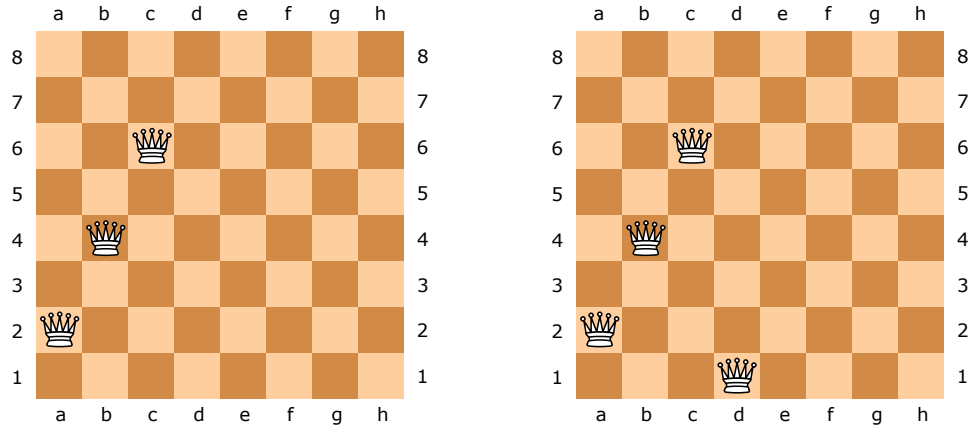
Die Lineare Suche ist systematisch. Der Suchraum hat deutlich weniger Elemente als der Suchraum bei der Zufallssuche: $8^8 = 16.777.216$, darunter aber alle Lösungen. Der Unterschied ist auf die Veränderung der Operatoren zurückzuführen, die nun einige ungültige Zustände gar nicht erst erzeugen.

Als weitere Verbesserung können wir das Problem so formulieren, dass nur noch Lösungskandidaten mit jeweils einer Dame pro Reihe und Spalte betrachtet werden, also Operatoren formulieren, die noch weniger ungültige Zustände erzeugt. Die Suchraumgröße beträgt dann noch $8! = 40.320$.

2.3.2 Teillösungen

Die bisher betrachteten Problemdefinitionen codieren nur Lösungskandidaten, die Verbesserungen wurden durch das Einbeziehen der Information aus der Problemspezifikation in die Suchraumkonstruktion erzielt. Als systematische Steuerungsstrategie kam nur das Aufzählen der Objekte in Frage. Die Codierung unterstützt keine weiteren Optionen.

Wir wollen auch Teillösungen codieren und erweitern die Codierung um das Symbol * für leere Felder (bzw. eine unbestimmte Platzierung). Das 8-Tupel (a2, *, *, *, *, *, *, *) repräsentiert eine Konfiguration mit nur einer Dame auf a2. Auf den ersten Blick bringt das keine Vorteile, denn der Suchraum vergrößert sich: Eine beliebige Anzahl von 1 bis 8 Damen verteilt auf die 64



(a) (a2, b4, c6, *, *, *, *, *)

(b) (a2, b4, c6, d1, *, *, *, *)

Abbildung 2.3: Zwei Teillösungen für das 8-Damen-Problem, 2.3(b) ist eine Verfeinerung von 2.3(a) mit einer weiteren Dame auf d1.

Felder ergibt $\frac{64!}{(64-8)!} \approx 1{,}78 \times 10^{14}$ Objekte im Suchraum. Darüber hinaus kann keines der hinzugekommenen Objekte eine Lösung sein, denn keines repräsentiert eine Stellung mit 8 Damen. Betrachten wir aber die Möglichkeiten der neuen Codierung. Wir codieren die Teilmenge aller Lösungskandidaten mit einer Dame auf a2 mit dem Tupel (a2, *, *, *, *, *, *, *). Durch Verfeinerung lassen sich die Teilmengen (a2, b1, *, *, *, *, *, *), (a2, b2, *, *, *, *, *, *) usw. bis (a2, b8, *, *, *, *, *, *) erzeugen. In diesem Beispiel wird eine Dame in Spalte b hinzugefügt. Aus der Problemspezifikation wissen wir, dass keine Lösungen mit zwei Damen in einer Reihe, in einer Spalte oder in einer Diagonale existieren. Die Teilmengen (a2, b1, *, *, *, *, *, *), (a2, b2, *, *, *, *, *, *) und (a2, b3, *, *, *, *, *, *) enthalten also keine Lösungen, es sind Sackgassen. Diese Mengen und damit große Teile des Suchraumes, nämlich alle Mengen, die eine der drei Teilmengen enthalten, müssen nicht weiter beachtet werden. Die Abbildung 2.3 zeigt eine Teillösung und eine verfeinerte Teillösung.

Anzumerken ist, dass die Codierung als Tupel sowohl eine Teillösung als auch das noch zu lösende Restproblem, also den Zustand, codiert. Die Teillösung, also der Pfad vom Startzustand zum aktuellen Zustand, ist die Reihenfolge in der die Damen gesetzt wurden. Der aktuelle Zustand ist direkt ersichtlich. Üblicherweise wird für den Zustand eine eigene Codierung eingeführt. Beim 8-Puzzle beispielsweise reicht die Sequenz der Verschiebungen bei bekanntem Startzustand zwar, um einen Zustand zu berechnen, dennoch wird die Position der Teile codiert, um Berechnungen zu vereinfachen.

Wir erstellen eine neue Formulierung für das 8-Damen-Problem, die mit

einem leeren Brett startet und Teillösungen verwendet:

- Startzustand : Ein leeres Schachbrett.
- Operatoren : Stelle eine Dame auf ein nicht attackiertes[5] Feld in der nächsten freien Spalte.
- Zieltest : Acht Damen auf dem Feld und keine kann eine andere schlagen.
- Pfadkosten : Die Kosten sind irrelevant.

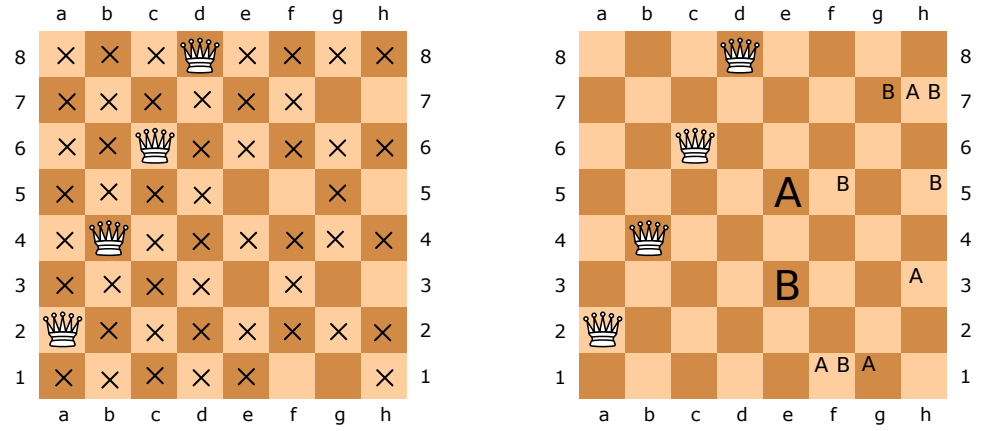
Mit dieser Problembeschreibung konstruieren wir einen Suchraum, der nur die laut Problemspezifikation zulässigen Stellungen mit 0 bis 8 Damen enthält, insgesamt 2.057 Objekte, darunter alle 92 Lösungen – tatsächlich sind alle erzeugten Stellungen mit 8 Damen auch Lösungen. Ein systematisches Verfahren könnte einen Operator anwenden bis 8 Damen auf dem Brett stehen. Wenn alle Felder in der nächsten noch unbesetzten Spalte attackiert sind, kann die Suche bei einem früheren Zustand mit weniger Damen wieder aufgenommen werden. Besonders in den ersten Schritten stehen jeweils mehrere Felder, das heißt mehrere Operatoren, zur Auswahl. Ein Suchverfahren, das so vorgeht, wird auch als *Backtracking* bezeichnet.

Die Auswahl zwischen möglichen Folgezuständen liefert einen Ansatz, um die Suche weiter zu verbessern. Um Sackgassen zu vermeiden, soll die nächste Dame beispielsweise so platziert werden, dass möglichst wenige neue Felder attackiert werden. Diese Idee können wir als Heuristik formulieren: Heuristik h = „Maximiere die Anzahl der nicht attackierten Felder“. Die Anzahl der nicht attackieren Felder wird als Bewertung der Situation nach einer möglichen Platzierung herangezogen. Ein informiertes Suchverfahren, das die Heuristik h nutzt, wird bei jedem Schritt so vorgehen, dass die meisten Optionen offen bleiben. Diese Idee wird auch als *Least Commitment* bezeichnet. Ein Beispiel zeigt Abbildung 2.4.

2.3.3 8-Puzzle

Betrachten wir eine Codierung für das 8-Puzzle. Die Operatoren sind Bewegungen $m \in \{up, right, down, left\}$. Lösungsobjekte sind Sequenzen $(m_i)_{i=1}^{N}$, $N \in \mathbb{N}$, die von einem beliebigen Startzustand s zum Zielzustand führen. Die Codierung der Bewegung ist eindeutig, denn wir betrachten die freie Stelle als „bewegt“. Das heißt, eine Bewegung nach rechts vertauscht die freie Stelle mit dem Teil rechts von ihr, das Teil wird demnach nach links verschoben. Eine

[5] Als attackiert bezeichnen wir Felder, auf denen eine Dame von einer schon vorhandenen geschlagen werden kann.



(a) (a2, b4, c6, d8, *, *, *, *)

(b) (a2, b4, c6, d8, {e3,e5}, *, *, *)

Abbildung 2.4: Auswahl im 8-Damen-Problem. Abb. 2.4(a) zeigt den aktuellen Zustand, attackierte Felder sind mit einem $\times$ markiert, 10 Felder sind unattackiert. Für die Spalte e stehen zwei Felder zur Wahl. Abb. 2.4(b) zeigt die Auswirkungen der jeweiligen Entscheidungen. Wird die nächste Dame auf e5 gesetzt (A), verbleiben vier unattackierte Felder (markiert mit einem kleinen A), wird die Dame auf e3 gesetzt (B) verbleiben fünf unattackierte Felder (markiert mit einem kleinen B).

Vorbedingung an die Operatoren ist, dass eine Bewegung nur durchgeführt werden kann, wenn sich auf der Zielposition auch ein Teil befindet, die freie Stelle darf also nicht aus dem Rahmen bewegt werden. Daraus ist ersichtlich, dass nicht in jedem Zustand auch alle Operatoren anwendbar sind. Eine formelle Beschreibung des Problems lautet:

- Startzustand: Eine beliebige (lösbare) Anordnung der 8 Teile.
- Operatoren: Bewegungen $m \in \{up, right, down, left\}$.
- Zieltest: Die Zielanordnung ist erreicht.
- Pfadkosten: Anzahl der Bewegungen.

Die Kenntnis des Startzustandes s und einer Teillösung $(m_k)_{k=1}^{N}, N \in \mathbb{N}$ reicht aus, um die Situation und damit auch den Zustand s_k, also die Position der 8 Teile und der freien Stelle, nach Ausführen der Züge zu beschreiben. Dennoch ist es sinnvoll den Zustand explizit zu codieren. Das 8-Puzzle zeigt, dass die Zustandsinformation nicht zur Beschreibung der Gesamtsituation ausreicht, denn ein Zustand kann auf verschiedenen Pfaden erreicht werden und

es ist möglich, einen Zustand durch eine bestimmte Sequenz in sich selbst zu überführen. Beispielsweise erzeugt die Sequenz $(left, right)$ den vorherigen Zustand. Der Zustandsraumgraph enthält also Kreise oder genereller: Redundante Pfade. Es ist nicht sinnvoll, mehr als einen Pfad zu einem bestimmten Zustand zu speichern. Die explizite Codierung des Zustandes vereinfacht nicht nur die Bewertung des Restproblems sondern auch die Erkennung redundanter Pfade.

In jedem Schritt des Suchprozesses stehen mehrere Züge zur Auswahl. Betrachten wir, wie der Suchprozesses gesteuert werden kann. Es scheint sinnvoll, den Abstand zum Zielzustand zu verringern, dafür müssen wir eine Idee des Abstandes formulieren. Eine Möglichkeit ist die Anzahl f_1 der Teile, die in einem bestimmten Zustand nicht in ihrer Zielposition sind. Eine zweite Möglichkeit ist die Summe f_2 der Manhattan-Distanzen über alle Teile hinsichtlich ihrer jeweiligen Zielposition. Die Manhattan-Distanz ist die minimal notwendige Anzahl der zulässigen Bewegungen, um ein Teil von einer Position in eine andere zu bewegen, wobei die anderen Teile ignoriert werden können. Befindet sich das Teil 5 beispielsweise in der Mitte der oberen Reihe, ist die Manhattan-Distanz für dieses Teil 1, befindet es sich in einer der Ecken, ist die Manhattan-Distanz 2. Zur Auswahl eines Zuges lassen sich entsprechend der genannten Abstandsformulierungen die folgenden Heuristiken formulieren: $h_1 =$ „minimiere die Anzahl der falsch positionierten Teile f_1“ beziehungsweise $h_2 =$ „minimiere die Summe der Manhattan-Distanzen f_2“.

Zu beachten ist, dass nur die möglichen Folgezustände miteinander verglichen werden, der aktuelle Zustand spielt keine Rolle. Die Gesamtsituation kann sich im Suchprozess also durchaus temporär verschlechtern. Eine systematische Suche muss darüber hinaus in der Lage sein, redundante Pfade zu erkennen. Zu diesem Zweck werden im Laufe des Suchprozesses die bereits bekannten Zustände gespeichert und neue mit den bekannten verglichen. Die dazu nötigen Mittel werden im folgenden Abschnitt eingeführt.

2.4 Zustandsraumrepräsentation

Wir betrachten eine formelle Beschreibung des Zustandsraumes. Wie in Abschnitt 2.1 bereits erwähnt, bildet der Zustandsraum einen Graph. Die Objekte des Zustandsraumes lassen sich als Knoten v darstellen. Ein Knoten repräsentiert also ein Restproblem (Zustand) und eine Teillösung (Pfad vom Startknoten zum Knoten v). Wie zuvor beschrieben ist ein Pfad eindeutig, während verschiedene Pfade zum gleichen Zustand führen können, das heißt verschiedene Knoten können den gleichen Zustand repräsentieren.

Die Graphstruktur ergibt sich, indem jeder Knoten mit seinem Vaterkno-

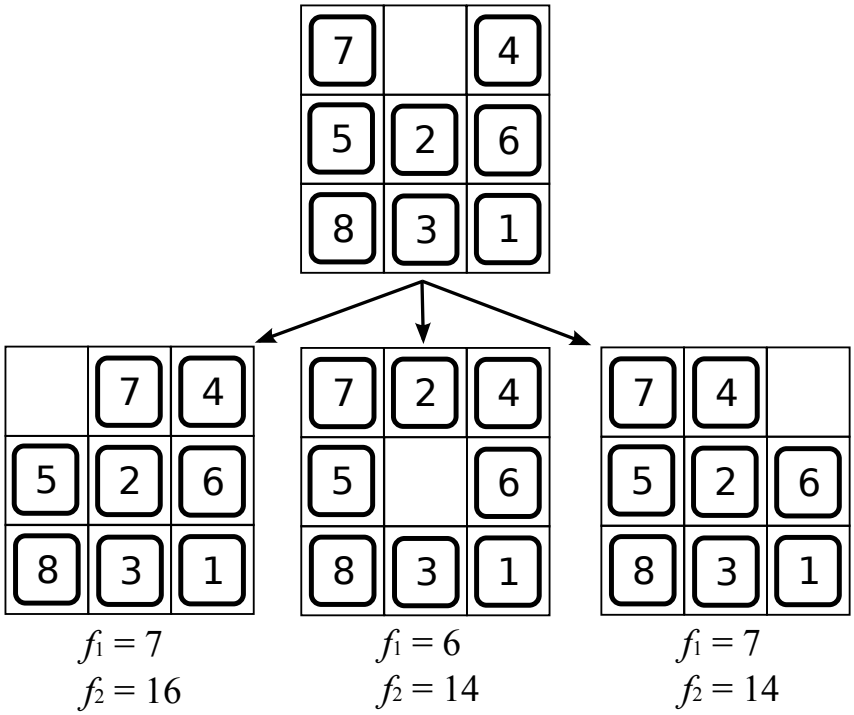


Abbildung 2.5: Auswahl im 8-Puzzle. Für jede der drei Möglichkeiten ist die Anzahl der falsch positionierten Teile (f_1) und die Summe der Manhattan-Distanzen für alle Teile (f_2) angegeben.

ten, also dem direkten Vorgänger, assoziiert wird. Die Anwendung eines Operators auf einen Knoten v erzeugt einen neuen Knoten, der als Kind von v bezeichnet wird. Dieser Prozess wird entsprechend Knotenerzeugung genannt. Man sagt auch v wird untersucht oder exploriert. Nur die direkten Nachfolger v' werden Kinder von v genannt, alle weiteren Nachfolger v'' haben keine spezielle Bezeichnung. Ebenso wird nur der direkte Vorgänger als Vater bezeichnet, alle weiteren einfach als Vorgänger.

Zwei wichtige Kenngrößen bei der Betrachtung eines Knotens im Zustandsraumgraphen sind der *Verzweigungsgrad* b (engl. *branching factor*) und die *Tiefe* d (engl. *depth*). Der Verzweigungsgrad b eines Knoten v gibt an, wie viele Kinder v hat und resultiert direkt aus der Anzahl der auf v anwendbaren Operatoren. Die Tiefe d gibt an, wie viele Schritte, das heißt Operatoranwendungen, zwischen dem Startknoten und v liegen.

Verfolgt man von einem beliebigen Knoten v die Kette der Vorgänger und die zugehörigen Operatoren bis zum Startknoten s erhält man einen eindeutigen Pfad. Wenn v einen Zielzustand repräsentiert, ist dieser Pfad eine Lösung des Problems. Ein Knoten v wird durch die folgenden Komponenten beschrie-

ben [RN10]:

- *Vater* : Der direkten Vorgängerknoten von v.
- *Operator* : Der Operator, der auf den Vater angewendet wurde, um v zu erzeugen.
- *Zustand* : Der Zustand, der von v repräsentiert wird.
- *Pfadkosten* : Die Kosten der Teillösung, die von v repräsentiert wird.

Die Knoten lassen sich nach ihrem Status im Prozess der Knotenerzeugung in vier Mengen unterteilen [Pea84]:

- *Expandiert* : Knoten, die vollständig untersucht sind; alle Kinder sind erzeugt.
- *Exploriert* : Knoten, die noch nicht vollständig untersucht sind; ein oder mehrere, aber nicht alle Kinder sind erzeugt.
- *Generiert* : Knoten, die erzeugt, aber noch nicht untersucht sind.
- *Unerforscht* : Knoten, die noch nicht erzeugt sind.

Aus der Menge der generierten Knoten wird im Suchprozess ein Knoten gewählt und untersucht. Die Knotenauswahl ist Teil der Suchstrategie. Die Menge der generierten Knoten bezeichnen wir mit OPEN.

Viele Suchverfahren benötigen auch Zugriff auf die expandierten Knoten, beispielsweise zur Erkennung von redundanten Pfaden. Die Menge der expandierten Knoten bezeichnen wir mit CLOSED.

2.5 Suchstrategien für Zustandsräume

Wir betrachten einige grundlegende Suchverfahren und ihre Eigenschaften, zuerst uninformierte, danach informierte Verfahren.

2.5.1 Tiefensuche

Tiefensuche ist eine uniformierte Suchstrategie, bei der jeweils der tiefste generierte Knoten zuerst untersucht wird. Gibt es mehrere generierte Knoten in maximaler Tiefe, wird unter diesen zufällig gewählt. Die Tiefensuche ist in Algorithmus 1 dargestellt.

Algorithmus 1 Tiefensuche

Input: s. Startknoten.
$successors(v)$. Liefert die Kinder von v.
$goal(v)$. Liefert *TRUE* wenn mit v ein Zielzustand erreicht ist, sonst *FALSE*.
Output: Eine Lösung oder das Symbol *FAILURE*.

```
if goal(s) then return s
put s on top of OPEN
loop
    if OPEN is empty then return FAILURE
    v ← top of OPEN
    remove v from OPEN
    put v to CLOSED
    foreach v' in successors(v) do
        if v' ∉ OPEN and v' ∉ CLOSED then
            if goal(v') then return v'
            put v' on top of OPEN
        end if
    end for
end loop
```

Bei der Tiefensuche ist OPEN ein Stapel, also eine Last-In-First-Out Struktur. Knoten werden nur oben in den Stapel eingefügt und nur von oben entfernt.

Die Knotenerzeugung ist in der Funktion $successors(v)$ realisiert, hier sind die Problemkomponenten $operators(v)$ und $result(o, v)$ (Vgl. Abschnitt 2.3) vereint. Die Reihenfolge, in der $successors(v)$ die Kindknoten liefert, ist unbestimmt.

Die Bedingung, dass ein Knoten nicht in OPEN und nicht in CLOSED ist, dient der Erkennung von redundanten Pfaden. Ist der Zustand eines neu generierten Knotens schon bekannt, so hat man diesen Zustand schon auf einem anderen Pfad erreicht. Der neue Knoten kann dann verworfen werden. Ein Beispiel, das den Sinn der Speicherung bekannter Knoten veranschaulicht, ist das 8-Puzzle (siehe Abschnitt 2.3.3). Die Anzahl der Zustände in diesem Problem ist endlich ($\frac{9!}{2} = 181.440$), jeder Zustand kann aber auf unendlich vielen Pfaden erreicht werden, wenn man redundante Pfade einbezieht.

Bei der Verwendung von Tiefensuche als Suchstrategie gilt es, bestimmte Eigenschaften zu beachten. Betrachten wir Suchräume mit unendlich langen Pfaden aber Lösungen in endlicher Tiefe, so kann Tiefensuche aufgrund der zufälligen Wahl eines Pfades bei mehreren Möglichkeiten in einen Pfad ohne Lösung gelangen und diesen unendlich lange verfolgen. Für solche Suchräume ist es sinnvoll, eine Maximaltiefe anzugeben. Ein Knoten, dessen Tiefe das Maximum überschreitet, wird verworfen.

Auch in endlichen Suchräumen kann der Speicheraufwand zu einem Problem werden, da zu jedem Zeitpunkt alle bekannten Knoten im Speicher gehalten werden. Für Pfade, in denen keine Lösung existiert, die also Sackgassen bilden, ist das unnötig. Eine Sackgasse liegt vor, wenn ein zur Expansion gewählter Knoten v keine Kinder hat oder alle Kinder die Tiefenschranke überschreitet. In diesem Fall können alle Vorgänger von v, die keine Nachfolger in OPEN haben, aus CLOSED entfernt werden. Diese Prozedur wird als „clean up CLOSED“ bezeichnet [Pea84]. Die Berechnung der entfernbaren Knoten ist aber sehr aufwändig, daher wird sie nur optional eingesetzt.

Tiefensuche ist ein systematisches Verfahren, um eine Lösung zu finden, die in endlicher Tiefe beziehungsweise unterhalb der Maximaltiefe liegt. Tiefensuche geht bei der Auswahl eines zu verfolgenden Pfades zufällig vor. Diese Auswahl intelligent zu treffen, ist der Ansatz von informierten Verfahren.

2.5.2 Breitensuche

Breitensuche ist eine uninformierte Suchstrategie, bei der zuerst alle Knoten in einer bestimmten Tiefe untersucht werden, bevor ein tieferer Knoten untersucht wird. Der Algorithmus ist schematisch genau so darstellbar wie Tiefensuche (Algorithmus 1), mit dem Unterschied dass OPEN eine Warteschlange ist, also eine First-In-First-Out Struktur. Knoten werden immer ans Ende von OPEN eingefügt und nur am Anfang entfernt.

Trotz der geringfügigen Änderung hat Breitensuche andere Eigenschaften als Tiefensuche. Auch in Suchräumen mit unendlich langen Pfaden werden Lösungen in endlicher Tiefe garantiert gefunden, ohne eine Maximaltiefe zu verwenden. Tatsächlich wird Breitensuche die Lösung mit der geringsten Tiefe zuerst finden, muss dabei aber alle bisher betrachteten Knoten im Speicher halten. Für Probleme wie das in Abschnitt 2.3.3 gezeigte 8-Damen-Problem, bei dem alle Lösungen in der gleichen Tiefe liegen, ist diese Eigenschaft ungünstig. Allgemein hat Breitensuche einen höheren Speicheraufwand als Tiefensuche. Vor allem in Suchräumen mit einem hohen durchschnittlichen Verzweigungsfaktor ist der Einsatz von Breitensuche problematisch.

2.5.3 Best-First-Suche

Bei informierten Strategien basiert die Auswahl eines Knotens v zur Expansion auf einer heuristischen Bewertungsfunktion $f(v)$. Diese Funktion kann verschiedene Faktoren einberechnen, beispielsweise Informationen über den Knoten v beziehungsweise seinen Zustand, Informationen über den Pfad vom Startknoten s nach v, Informationen aus der Problem- und Zielbeschreibung sowie Wissen aus der Problemdomäne.

Algorithmus 2 Best-First Suche

Input: s. Startknoten.
$successors(v)$. Liefert die Kinder von v.
$goal(v)$. Liefert *TRUE* wenn mit v ein Zielzustand erreicht ist, sonst *FALSE*.
$f(v)$. Evaluierungsfunktion für einen Knoten v.
Output: Ein Zielknoten oder das Symbol *FAILURE*.

```
put s to OPEN
if goal(s) then return s
loop
    if OPEN is empty then return FAILURE
    v ← a node from OPEN that minimizes f
    remove v from OPEN
    put v to CLOSED
    foreach v' in successors(v) do
        if goal(v') then return v'
        if v' ∉ OPEN and v' ∉ CLOSED then
            put v' to OPEN
        else
            v_old ← a node from OPEN or CLOSED that equals v'
            if f(v') < f(v_old) then
                remove v_old from OPEN or CLOSED
                put v' to OPEN
            end if
        end if
    end for
end loop
```

Die Suchstrategie wird entscheidend von der Bewertungsfunktion f bestimmt. Ansonsten sind sich die Verfahren sehr ähnlich, eine generelle Schematik ist die *Best-First-Suche*: Der zu expandierende Knoten wird unter allen Knoten in OPEN als derjenige gewählt, der f minimiert. Die Funktion f wird dabei als Kostenabschätzung betrachtet (es ist auch möglich, den Nutzen abzuschätzen, in diesem Fall würde f maximiert). Führen mehrere Pfade zu einem Knoten, wird nur der gemerkt, der zum minimalen f gehört, alle anderen werden verworfen. Die Best-First-Suche ist in Algorithmus 2 dargestellt. Im folgenden Abschnitt wird eine spezielle Version der Best-First-Suche betrachtet.

2.5.4 Die A*-Suche

Betrachten wir einzelne Komponenten von $f(v)$. Sei $h(v)$ eine heuristische Funktion zur Aufwandsabschätzung für das Restproblem. Die Heuristik $h(v)$ liefert also eine Schätzung für den günstigsten Pfad von v zu einem Lösungszustand. Diese Funktion kann Wissen über die Problemdomäne beinhalten. Bei-

spiele sind die in Abschnitt 2.3.3 genannten Heuristiken, die Anzahl der Teile die nicht in ihrer Zielposition sind und die Summe der Manhattan-Distanzen. Sei $g(v)$ eine Funktion, die die Kosten des Pfades vom Startknoten nach v liefert. Kombinieren wir für einen Knoten v die Abschätzung $h(v)$ mit den Pfadkosten $g(v)$ zu $f(v) = g(v) + h(v)$, so schätzt $f(v)$ den Aufwand einer Lösung mit einem Pfad durch v ab.

Als *optimale* Lösung wird eine Lösung mit den geringsten Kosten bezeichnet. Wir können zeigen, dass eine Bestensuche eine optimale Lösung finden kann. Der Zieltest muss dafür nicht nach der Generierung sondern nach der Auswahl eines Knotens zur Expandierung durchgeführt werden. Die Funktionen g und h müssen bestimmte Anforderungen erfüllen, die im Folgenden besprochen werden (vergleiche [RN10]).

Für die Pfadkosten sollen die in Abschnitt 2.3 gezeigten Annahmen gelten: Die Funktion $g(v)$ liefert die Summe der nicht negativen Schrittkosten $cost(v, v')$ auf dem Pfad vom Startknoten nach v.

Für h nehmen wir an, dass der Wert nicht negativ ist. Für einen Zielknoten v sei $h(v) = 0$. Außerdem soll h *konsistent* sein, das heißt für beliebige v und Nachfolger v' soll gelten $h(v) \leq cost(v, v') + h(v')$. Eine konsistente Heuristik ist immer auch *zulässig*, das heißt, dass $h(v)$ die tatsächlichen Kosten für einen Pfad von v zu einer Lösung *nicht überschätzt*. Die Beispielheuristiken für das 8-Puzzle in Abschnitt 2.3.3 sind zulässig und konsistent.

Aus der Konsistenz folgt, dass die Werte für f entlang eines Pfades monoton steigen, denn für alle Nachfolger v' eines Knotens v gilt:

$$f(v') = g(v') + h(v') = g(v) + c(v, v') + h(v') \geq g(v) + h(v) = f(v).$$

Für einen Knoten v in OPEN, der f minimiert, also vom Suchverfahren zur Expansion gewählt wird, wurde der optimale Pfad vom Startknoten s nach v gefunden. Das lässt sich durch Ausschluss zeigen: Angenommen ein Knoten v' liegt auf dem optimalen Pfad vor v. Da f entlang eines Pfades monoton steigend ist, würde v' zuvor gewählt.

Da für einen zur Expansion ausgewählten Knoten v der optimale Pfad gefunden wurde, folgt, dass wenn v ein Lösungsknoten ist auch eine optimale Lösung gefunden wurde.

Eine Best-First-Suche, die eine Bewertungsfunktion mit diesen Eigenschaften verwendet ist die *A*-Suche* [HNR68, HNR72], dargestellt in Algorithmus 3.

2.6 Anforderungen in der Praxis

Die im Abschnitt 2.5.4 vorgestellte A*-Suche ist effizient und findet eine optimale Lösung. Im praktischen Einsatz gibt es mehrere Herausforderungen, von

Algorithmus 3 A*-Suche

Input: s. Startknoten.
$successors(v)$. Liefert die Kinder von v.
$goal(v)$. Liefert *TRUE*, v ein Zielzustand ist, sonst *FALSE*.
$f(v)$. Evaluierungsfunktion für einen Knoten v.
$g(v)$. Liefert die Pfadkosten für einen Knoten v.
Output: Ein Zielknoten oder das Symbol *FAILURE*.

```
put s to OPEN
if goal(s) then return s
loop
    if OPEN is empty then return FAILURE
    n ← a node from OPEN that minimizes f
    if goal(v) then return v
    remove v from OPEN
    put v to CLOSED
    foreach v' in successors(v) do
        if v' ∉ OPEN and v' ∉ CLOSED then
            put v' to OPEN
        else
            v_old ← a node from OPEN or CLOSED that equals v'
            if g(v') < g(v_old) then
                remove v_old from OPEN or CLOSED
                put v' to OPEN
            end if
        end if
    end for
end loop
```

denen zwei in diesem Abschnitt angesprochen werden sollen: Die Formulierung einer Heuristik und der Umgang mit begrenztem Speicher.

2.6.1 Entwicklung einer Heuristik

Für den praktischen Einsatz eines informierten Suchverfahrens ist die Entwicklung einer guten Heuristik eines der wichtigsten Teilprobleme. In manchen Fällen lässt sich eine Heuristik durch Vereinfachen des Problems entwickeln. Ein Beispiel hierfür ist Anwendung der Manhattan-Distanz beim 8-Puzzle (vgl. Abschnitt 2.3.3). Die Berechnung vereinfacht das Problem, indem sie davon ausgeht, dass jedes Teil mit einer minimalen Anzahl Züge in seine Zielposition verschoben werden kann, ohne die anderen Teile zu beachten, die den Weg versperren. Formell gesprochen wird eine Vorbedingung an die Operatoren weggelassen, man bezeichnet diese Vorgehensweise auch als *Relaxation*.

Eine weitere Möglichkeit ist, Techniken des Maschinellen Lernens zu verwenden um die Abschätzung für Zustände zu erlernen. Mit dieser Methode

wird die Heuristik sozusagen trainiert.

Eine dritte Möglichkeit ist, Datenbanken mit trivialen Teilproblemen, für die Lösungen bekannt sind, zu verwenden. Beim 8-Puzzle lässt sich beispielsweise für jede Stellung der Teile der optimale Zug notieren, dieses Problem gilt damit als komplett gelöst [Rei93]. Für n-Puzzle-Versionen mit mehr als acht Teilen, beispielsweise das 15-Puzzle, ist solch eine Datenbank wegen der kombinatorisch ansteigenden Anzahl der Stellungen mit der aktuellen Technik praktisch nicht möglich. Ein Beispiel für den Einsatz von Teilproblemen sind Endspieldatenbanken für Schach, die für alle möglichen Stellungen mit wenigen Figuren die optimalen Züge notieren. Eine ganze Schachpartie vorzuberechnen ist dagegen praktisch unmöglich.

2.6.2 Umgang mit begrenztem Speicher

Alle Suchalgorithmen, die auf Graphen operieren – das schließt alle in den vorhergehenden Abschnitten vorgestellten Verfahren ein – müssen alle generierten Knoten im Speicher halten. Dies ist für große Suchräume in der Praxis problematisch, da nur begrenzter Speicher zur Verfügung steht. Eine Anforderung an die Codierung der Knoten ist daher, so speichereffizient wie möglich zu sein. Von den Suchverfahren wurden verschiedene Spezialisierungen entwickelt, die mit dem verfügbaren Speicher auskommen ohne die günstigen Eigenschaften, vor allem die Optimalität, aufzugeben. Eine übliche Strategie ist, bei knappem Speicher den am schlechtesten bewerteten Knoten zu verwerfen. Dabei wird die Bewertung des verworfenen Knotens an dessen Vater übergeben um die Information zu erhalten. Dieser Schritt wird als *backup* bezeichnet. Eine solche Strategie ist beispielsweise im Algorithmus *Simplified Memory-Bound A* (SMA*)* formuliert [Rus92]. Der backup-Schritt kann jedoch zu inkonsistenten Bewertungen führen, was weitere Modifikationen erforderlich macht, wenn optimales Verhalten gewährleistet bleiben soll [ZH02].

In vielen Fällen ist es nicht nötig, einen optimalen Pfad zu finden. Wenn das Problem beispielsweise als Bedingungserfüllungsproblem formuliert ist, reicht es aus, eine Lösung zu finden, die die Bedingungen erfüllt.

Wir betrachten die Konstruktion eines Akrostichons als Bedingungserfüllungsproblem, zumal nicht klar ist, wie eine „optimale“ Lösung aussehen sollte. Wir verwenden Heuristiken zur Minimierung des Suchaufwandes.

Kapitel 3

Operatoren

Unser Ziel ist die Erzeugung eines vorgegebenen Akrostichons in einem vorgegebenen Text. Wir betrachten diese Aufgabe als Suchproblem. Im vorherigen Kapitel wurden Suchverfahren für Zustandsräume vorgestellt. Nun müssen wir Methoden entwickeln, mit denen wir aus einem vorgegebenen Text einen Suchraum verschiedener Textvarianten erstellen können, also Methoden zur Textveränderung. Diese Methoden setzen wir dann als Operatoren in einem Suchverfahren ein.

In diesem Kapitel diskutieren wir die Operatoren. Wir betrachten, welche Möglichkeiten zur Veränderung von Texten bestehen, unter welchen Bedingungen sie angewendet werden können und welche Auswirkungen sie haben. Wir diskutieren auch Kriterien, nach denen wir die Operatoren bewerten können, beispielsweise um Operatorkosten (vgl. Abschnitt 2.3) festzulegen.

3.1 Möglichkeiten zur Textveränderung

Die Möglichkeiten zur Veränderung eines Textes lassen sich grob in vier Kategorien einteilen: Einfügen, Entfernen, Ersetzen und Vertauschen. Betrachten wir einige allgemeine Beispiele für jede Kategorie.

Wenn wir als Ausgangszustand einen Text ohne Zeilenumbruch- oder Absatzmarkierungen annehmen, dann ist das Einfügen von Zeilenumbrüchen notwendig, um ein Akrostichon zu erstellen – zumindest wenn das Akrostichon länger als ein Buchstabe ist. Sind in dem Text schon Zeilenumbrüche vorhanden, so ist es wahrscheinlich, dass sich diese nicht an den richtigen Positionen befinden. Wir brauchen in diesem Fall also Operatoren um die Zeilenumbruchpositionen zu vertauschen und zusätzlich Zeilenumbrüche zu entfernen oder einzufügen, wenn die Anzahl der vorhandenen Zeilenumbrüche nicht mit der Länge des gewünschten Akrostichons übereinstimmt.

Wir können auch Wörter, Wortfolgen oder ganzen Sätze in den Text einfügen

oder aus dem Text entfernen. Wir bezeichnen Wortfolgen als *Phrasen*, um sie von einzelnen Wörtern zu unterscheiden, da wir in der detaillierten Betrachtung der Operatoren auch Wort-Operatoren von Phrasen-Operatoren unterscheiden wollen.

In die Kategorie „Ersetzen“ fallen beispielsweise Operatoren, die Wörter durch Synonyme ersetzen. Auch die Flexion, das heißt Veränderung grammatischer Eigenschaften wie Zeitform, Zahl oder Fall kann als Ersetzung betrachtet werden, teilweise werden dabei aber auch Wörter eingefügt oder ihre Position vertauscht. Vertauschen lassen sich weiterhin Elemente von Aufzählungen oder Sätze die in keinem Kausalzusammenhang stehen.

Einige dieser Möglichkeiten diskutieren wir in diesem Kapitel im Detail. Um die Idee der Akrostichonerzeugung zu demonstrieren haben wir zunächst nur einen Teil der Operatoren implementiert. Unser System ist darauf ausgelegt, dass neue Operatoren mit wenig Aufwand integriert werden können. Eine Reihe Operatoren, die noch nicht implementiert wurden, werden in einem gesonderten Abschnitt diskutiert. Wir betrachten die genannte Operatoren nicht als ausreichend um jedes beliebige Akrostichon in jedem beliebigen Text erzeugen zu können, denn in einigen Fällen wird der erzeugbare Suchraum keine Lösungen enthalten. Je mehr Operatoren wir zur Verfügung haben, desto umfangreicher wird der Suchraum und desto höher ist die Wahrscheinlichkeit, dass eine Lösung existiert. Deswegen muss die Möglichkeit bestehen, weitere Operatoren hinzuzufügen. Wir nehmen an, dass wir mit mehr Operatoren sowohl mehr, als auch „bessere“ Lösungen finden können, wobei noch zu diskutieren ist, wie die Qualität einer Lösung eingeschätzt werden kann.

3.2 Bewertungskriterien für Operatoren

Hinsichtlich der Qualität einer Lösung betrachten wir vier Aspekte des Textes: Formatierung, Rechtschreibung, Grammatik und Semantik. Auf dieser Grundlage wollen wir die Operatoren bewerten, das heißt wir betrachten inwiefern die Operatoren die Qualität einer Lösung in diesen Aspekten beeinflussen. Unser Hauptziel ist jedoch die Erzeugung eines Akrostichons und diesem Ziel werden die Qualitätskriterien untergeordnet. Hier können wir noch ein weiteres Bewertungskriterium einführen: Die Stärke eines Operators. Wir betrachten die Kriterien in den folgenden Absätzen.

Formatierung Dies ist ein optisches Kriterium, es betrifft vor allem die Längen der einzelnen Zeilen im Text. Normaler Text ist entweder beidseitig bündig (Blocksatz) oder hat leicht unterschiedliche Zeilenlängen (Flattersatz). Wir erwarten nicht, optisch perfekte Texte erzeugen zu können. Die Erstellung

möglichst einheitlicher Zeilenlängen soll aber als Richtlinie dienen, um die Operatoren, die die Formatierung beeinflussen, also die Zeilenumbruchoperatoren, zu bewerten oder aus verschiedenen Lösungen zu wählen.

Rechtschreibung Die korrekte Schreibweise als Kriterium zur Bewertung heranzuziehen ist vielleicht auf den ersten Blick nicht sinnvoll, wir entwickeln aber auch Operatoren, die bewusst Rechtschreiberegeln verletzen, wenn sie damit viel zur Konstruktion eines Akrostichons beitragen können. Außerdem können wir auch nicht ausschließen, dass der Eingabetext bereits Fehler enthält. Wenn eine Wahlmöglichkeit besteht, sollen natürlich Operatoren bevorzugt werden, die keine Rechtschreibfehler erzeugen.

Grammatik Alle Operatoren, die Wörter oder Phrasen einfügen, entfernen, ersetzen oder vertauschen, können damit die grammatische Struktur eines Satzes beschädigen. Um dies festzustellen, könnten wir Werkzeuge des Natural Language Processing (NLP) zur Strukturanalyse von Texten (Phrasenstrukturgrammatikparser, POS-Tagger, Chunker, ...) verwenden. Diese Werkzeuge basieren auf statistischen Modellen und sind sehr weit entwickelt. Da die Natürliche Sprache selbst aber keinen festen Regeln unterworfen ist, lässt sich auch mit den besten Werkzeugen nur mit einer gewissen Wahrscheinlichkeit feststellen, ob ein Satz grammatisch richtig ist. Wir können also nicht als Bedingung formulieren, dass alle Operatoren nur grammatisch korrekte Ergebnisse liefern müssen. Wir können aber das Potenzial der Operatoren in dieser Hinsicht einschätzen und Operatoren mit geringerer Neigung zu grammatischen Fehlern bevorzugen, wenn wir die Wahl haben.

Semantik Die größte Herausforderung bei der maschinellen Veränderung von Text besteht darin, mit der Veränderung nicht den Sinnzusammenhalt zu zerstören. Für einen Menschen ist das kein Problem, Computer sind jedoch nicht einmal in der Lage, zu erkennen, ob ein Text überhaupt Sinn ergibt. Dementsprechend können wir auch nicht feststellen, ob eine Änderung den Sinn verfälscht. Auch in dieser Kategorie können wir nur das Fehlerpotenzial der Operatoren einschätzen.

Stärke eines Operator Als Stärke eines Operators bezeichnen wir sein Potenzial, zu einer Lösung mit möglichst geringem Gesamtaufwand beizutragen. Den Aufwand für eine Lösung können wir durch die verwendeten Ressourcen, das heißt Speicher und Zeit, oder generell durch die Länge des Lösungspfades beziffern. In unserem Fall sind starke Operatoren solche, die häufig anwendbar sind und viele verschiedene Buchstaben für die Erzeugung des Akrostichons bereitstellen können.

Ideal wäre es, starke Operatoren zu haben, die wenig bis gar kein Risiko für die Qualität der Lösung bedeuten. In der Praxis sind unsere stärksten Operatoren leider auch sehr risikobehaftet. Das Mittel zur Abwägung zwischen aufwendigen und qualitativ besseren und weniger aufwendigen, dafür schlechteren Lösungen sind die Operatorkosten (vorausgesetzt wir verwenden ein Suchverfahren bei dem die Operatorkosten zur Steuerung des Suchprozesses herangezogen werden).

In unserem Demonstrationssystem verwenden wir einheitliche Kosten für alle Operatoren und nehmen somit Fehler in allen Qualitätskategorien in Kauf, wenn damit ein schneller Fortschritt im Suchprozess zu erzielen ist. In der Betrachtung der Operatoren diskutieren wir jedoch auch ihre Stärke und ihren Einfluss auf die Lösungsqualität. Semantik und Grammatik sind uns dabei wichtiger als Rechtschreibung und Formatierung. Semantik und Grammatik sind allerdings auch die Quatitätskategorien, in denen Fehler schwerer zu erkennen und zu vermeiden sind.

3.3 Operatortypen und Operatornotation

Die Anwendung eines Operators beschreibt eine Veränderung des Textes, also eine Zustandsänderung. In Abschnitt 2.4 wurde beschrieben, dass eine Sequenz von Zustandsänderungen, das heißt der Pfad durch den Zustandsraumgraphen, eindeutig codiert sein muss. Wir müssen uns also bei der Diskussion der Operatoren Gedanken darüber machen, wie diese eindeutige Codierung herstellbar ist. Betrachten wir dazu beispielsweise die Anwendung eines Synonymoperators. Wir können diese Operatoranwendung so beschreiben, das für jedes Wort des Textes eine Menge Synonyme erzeugt werden und der Operator somit mehrere Folgezustände erzeugt. Für einen Text aus drei Wörtern, bei dem für jedes Wort zwei mögliche Synonyme erzeugt werden, erzeugt eine Synonymoperatoranwendung so betrachtet also sechs Nachfolgezustände. Um den Pfad von einem Textzustand zu einem Nachfolger eindeutig zu beschreiben, müssen wir aber codieren, welches Wort durch welches Synonym ersetzt wurde. Wir können das Beispiel also auch so betrachten, dass auf einen Textzustand sechs verschiedene Synonymoperatoren angewendet werden, um jeweils genau einen Nachfolgezustand zu erzeugen. Die beiden Betrachtungsweisen sind hinsichtlich des Suchvorganges analog.

Wir unterscheiden konzeptionell zwischen dem *Operatortyp* (beispielsweise einem Synonymoperator, der mehrere Nachfolgezustände erzeugt) und einem *speziellen Operator* eines Typs (beispielsweise einem Synonymoperator, der genau einen Nachfolgezustand erzeugt). In den folgenden Abschnitten benutzen wir den Begriff „Operator“ für beide Konzepte. Wenn wir eine bestimmte

Änderung im Text beschreiben und Überlegungen anstellen, welche Informationen zur eindeutigen Codierung eines Operators notiert werden müssen, ist ein spezieller Operator gemeint. Bei der Beurteilung beziehen wir uns hingegen auf den Typ, denn alle Operatoren eines Typs haben in der Regel die gleichen Eigenschaften hinsichtlich der in Abschnitt 3.2 genannten Kriterien (Formatierung, Rechtschreibung, Grammatik, Semantik, Stärke). Teilweise lassen sich aber auch Parameter festlegen, die die Eigenschaften beeinflussen. Bei den Synonymoperatoren spielt beispielsweise der Kontext eines zu ersetzenden Wortes eine Rolle, und ein Parameter bestimmt, wieviel Kontext betrachtet wird. Damit ändert sich zum einen die Wahrscheinlichkeit, dass Fehler auftreten (hier vor allem die hinsichtlich des Semantik-Kriteriums), zum anderen die Anzahl der Synonymkandidaten (also die Stärke). Je mehr Kontext wir verwenden, desto weniger wahrscheinlich ist ein Fehler aber auch die Anzahl der Kandidaten sinkt (genaueres in Abschnitt 3.6.1). Operatoren mit Parametern können als Unterklassen eines Typs betrachtet werden.

3.4 Trennoperatoren

Wir gehen davon aus, dass der Ausgangszustand ein unformatierter Text ist. Das heißt im Besonderen wir gehen von einem Text ohne Zeilenumbrüche aus. Eine naheliegende Vorgehensweise ist dann, den Text durch Einfügen von Zeilenumbrüchen so zu formatieren, dass ein Akrostichon entsteht. Dieser Abschnitt behandelt Operatoren, die Zeilenumbrüche erzeugen. Wir unterscheiden dabei zwischen Zeilenumbrüchen an Wortgrenzen (die wir allgemein als Zeilenumbruch bezeichnen), Zeilenumbrüchen innerhalb von Worten (Silbentrennung) und Zeilenumbrüchen zwischen Sätzen (die auch als Absatz interpretiert werden können).

3.4.1 Zeilenumbruchoperator

Betrachten wir ein einfaches Beispiel. Der folgende Satz beschreibt ein *hello world*-Programm:

```
Hello world is an easy example program to learn a new programming
language before writing other, more complex programs.
```

Wir betrachten jeden Text so, dass er in jedem Zustand ein Akrostichon enthält, nämlich die Buchstaben am Zeilenbeginn, und bezeichnen dieses als *aktuelles Akrostichon*. Im Beispiel (angenommen der Text besteht aus einer Zeile und der Umbruch ist nur der Darstellung geschuldet) ist das aktuelle Akrostichon nur einen Buchstaben lang: „H“. Wir wollen in diesem Text das

Wort „Hello“ als *Zielakrostichon* erzeugen. Das Beispiel ist so konstruiert, dass es ausreicht, den Text an bestimmten Positionen umzubrechen:

```
Hello world is an easy
example program to
learn a new programming
language before writing
other, more complex programs.
```

Die Lösung entsteht durch die mehrfache Anwendung eines Zeilenumbruchoperators. Wie in Abschnitt 3.3 beschrieben, können wir die Lösung auch als Anwendung von vier speziellen Zeilenumbruchoperatoren betrachten. Die Aufgabe im Suchprozess ist es, die vier richtigen Operatoren in der richtigen Reihenfolge zu wählen. Ein Zeilenumbruchoperator unterliegt gewissen Vorbedingungen: Er kann nur vor Wörtern eingesetzt werden und muss dort ein Leerzeichen ersetzen. Diese Bedingungen sollen beispielsweise verhindern, dass zwischen einem Wort und einem Satzzeichen getrennt wird.

Wir führen eine Notation für einen speziellen Zeilenumbruchoperator ein: $\textsc{Lb}(i)$ bezeichnet einen Zeilenumbruch (engl. linebreak) vor dem i-ten Wort im Text. Mehr Information als die Position im Text, an der umgebrochen wird, ist im Fall des Zeilenumbruchs nicht nötig. Die sukzessive Anwendung des Operators auf den Text notieren wir als Sequenz $[\textsc{Lb}(i), \textsc{Lb}(j), \ldots]$. Die Lösung des Beispiels lautet in dieser Notation $[\textsc{Lb}(6), \textsc{Lb}(9), \textsc{Lb}(13), \textsc{Lb}(16)]$.

Betrachten wir den Suchraum, der allein mit Zeilenumbruchoperatoren erzeugt werden kann und entwickeln eine Suchstrategie. Im Startzustand stehen 18 spezielle Operatoren zur Wahl. Der Text besteht zwar aus 19 Wörtern aber vor dem ersten Wort darf laut Vorbedingung nicht umgebrochen werden. Nach der Anwendung eines Operators bleiben 17, dann 16 und schließlich 15 Möglichkeiten. Mehr als vier Zeilenumbrüche einzufügen würde den Suchraum nur unnötig vergrößern, denn das Akrostichon „Hello“ besteht nur aus fünf Buchstaben und der erste Buchstabe muss (so haben wir es in Abschnitt 1.2 festgelegt) gleichzeitig der erste Buchstabe des Textes sein. Wir benötigen also maximal vier Zeilenumbrüche und jeder Pfad, auf dem mehr Umbrüche eingefügt werden, ist eine Sackgasse. Der Suchraum besteht aus $18 \times 17 \times 16 \times 15 = 73.440$ Knoten. Das scheint nicht viel, schon gar nicht für die Rechenleistung aktueller Computer, die problemlos in der Lage wären, alle diese Knoten aufzuzählen. Die Suchraumgröße steigt jedoch exponentiell mit der Anzahl der Wörter im Text. Dies ist eine ungünstige, für Suchprobleme aber typische Eigenschaft. Betrachten wir als Praxisbeispiel den Schwarzenegger-Brief (Abbildung 1.1 auf Seite 5). Der Text mit dem Akrostichon besteht aus 85 Wörtern. Die gleiche Rechnung basierend auf 85 statt 18 Wörtern ergibt über 48 Millionen Knoten. Während die Anzahl der Wörter auf weniger als

das fünffache steigt, steigt die Anzahl der Knoten auf das 650-fache. Dabei ist ist zu beachten, dass wir gerade nur Operatoren eines Typs (Zeilenumbruch) betrachten. Mit Hinzunahme weiterer Operatoren potenzieren sich die Möglichkeiten. Die Operatoren sinnvoll einzusetzen ist also notwendig, um den Suchraum unter Kontrolle zu behalten, auch wenn wir prinzipiell nichts an der exponentiellen Natur des Problems ändern können.

Ein erster Ansatzpunkt zur Optimierung ist, dass viele der Knoten gleiche Zustände referenzieren. Beispielsweise erzeugt die Sequenz [LB(2), LB(3)] den gleichen Zustand wie [LB(3), LB(2)]; nämlich jeweils einen Zeilenumbruch vor „world“ und vor „is“. Die Anzahl der verschiedenen Zustände ist deutlich geringer als die Anzahl der Knoten, sie berechnet sich als Summe der Kombinationen von 0 (Startzustand) bis 4 Zeilenumbrüche auf 18 Positionen: $\sum_{k=0}^{4} \binom{18}{k} = 4048$. Suchverfahren für Zustandsräume, die alle bekannten Zustände speichern (vergleiche Abschnitt 2.5), sind in der Lage, äquivalente Zustände zu erkennen und speichern nur jeweils einen Pfad, der zu einem bestimmten Zustand führt, das heißt es wird nur einer der vielen Knoten gespeichert, die einen bestimmten Zustand referenzieren. Wir können jedoch in diesem Fall die Operatoren auch so gestalten, dass schon die Erzeugung redundanter Knoten verhindert wird.

Dazu formulieren wir eine weitere Bedingung für den Einsatz eines speziellen Zeilenumbruchoperators: Eine einmal erstellte Zeile soll entlang des gleichen Pfades nicht erneut umgebrochen werden. Neue Zeilenumbrüche werden nur nach den vorhandenen eingefügt. Diese Vorbedingung an einen Zeilenumbruchoperator lautet formell: Die Position für einen Zeilenumbruch muss größer als alle auf dem gleichen Pfad vorhandenen Zeilenumbruchpositionen sein. Im Startzustand stehen damit weiterhin 18 Positionen zur Auswahl, für den zweiten Schritt sind es 17, falls im ersten Schritt LB(2) angewendet wurde, aber nur eine falls LB(18) angewendet wurde. Wurde LB(19) angewendet, kann kein weiterer Zeilenumbruchoperator angewendet werden, ohne die Bedingung zu verletzen, dies ist also eine Sackgasse. Ebenso führt die Wahl von LB(18) in eine Sackgasse, weil nur ein weiterer Operator, nämlich LB(19), auf diesem Pfad anwendbar ist, zur Lösung aber mindestens vier Zeilenumbrüche nötig sind.

3.4.2 Zeilenlänge

Im Beispiel existiert noch eine weitere Möglichkeit das Akrostichon zu erzeugen. Der erste Zeilenumbruch kann auch vor dem Wort „easy“ statt vor „example“ eingefügt werden, um das „e“ für „Hello“ zu erzeugen:

```
Hello world is an
easy example program to
learn a new programming
language before writing
other, more complex programs.
```

Beide Varianten erfüllen die Zielbedingung, sie enthalten „Hello“ als Akrostichon. Wenn mehrere Lösungen gefunden werden, ist es wünschenswert, die Qualität beurteilen zu können. In Abschnitt 3.2 haben wir die Formatierung als ein Kriterium genannt. Dieses können wir hier verwenden. Dazu betrachten wir den Unterschied der Zeilenlängen und nehmen an, dass Texte mit wenig Unterschied in den Zeilenlängen optisch ansprechender sind. Wir definieren die Zeilenlänge l als Anzahl der Zeichen in der Zeile und betrachten die Menge aller Zeilenlängen als Zufallsvariable L, wobei jedes Element mit gleicher Wahrscheinlichkeit $p(l) = \frac{1}{n}$ mit n als Anzahl der Zeilen auftritt. Eine Bewertung des Unterschieds in den Zeilenlängen liefert die Standardabweichung σ. Um σ zu berechnen, müssen wir den Erwartungswert E und die Varianz Var von L bestimmen. Wir verwenden wir den arithmetischen Mittelwert für E und bestimmen Var und σ wie folgt [MS99]:

$$\begin{aligned} E(L) &= \frac{1}{n}\sum_{l\in L} l \\ Var(L) &= \frac{1}{n}\sum_{l}(l - E(L))^2 \\ \sigma &= \sqrt{Var(L)} \end{aligned}$$

Wir betrachten dabei nur die im Suchvorgang erzeugten Zeilen, der Text nach dem letzten eingefügten Zeilenumbruch wird ignoriert (im Beispiel wäre das die letzte Zeile, die mit „other“ beginnt). Die erste Beispiellösung, mit dem Zeilenumbruch vor „example“, hat die Zeilenlängen $22, 18, 23, 23$, und damit $\sigma = 2{,}06$, die zweite Beispiellösung mit dem Umbruch vor „easy“ hat die Zeilenlängen $17, 23, 23, 23$ und damit $\sigma = 2{,}6$. Die erste Lösung hat also einheitlichere Zeilenlängen und wird mit dieser Bewertungsmethode als besser eingeschätzt.

Eine solche Bewertung ist nur nötig, wenn mehrere Lösungen gegeneinander abzuwägen sind. Dies ist beispielsweise der Fall, wenn wir mehrere Suchvorgänge mit unterschiedlichen Zeilenlängenvorgaben durchführen, wie in Abschnitt 5.2 beschrieben. Die Zeilenlängen, das heißt die Formatierung sollte auch nicht das einzige Kriterium sein, wir wollen auch den Konstruktionsprozess, beispielsweise hinsichtlich der möglicherweise eingebauten Fehler mit in die Bewertung einbeziehen.

Das Formatierungskriterium, das heißt eine Lösung mit möglichst einheitlichen Zeilenlängen zu erzeugen, lässt sich aber auch schon bei der Anwendung der Zeilenumbruchoperatoren beachten, indem wir fordern, dass Zeilenumbrüche in möglichst gleichmäßigen Abständen eingefügt werden. Es ist

auch sinnlos, in den ersten Schritten der Suche Umbrüche am Ende des Textes einzufügen, denn das führt mit hoher Wahrscheinlichkeit in Sackgassen.

Wir erweitern deswegen die oben eingeführte Vorbedingung, dass neue Zeilenumbrüche nur nach allen vorhandenen eingefügt werden dürfen, um die Bedingung, dass ein neu einzufügender Zeilenumbruch in einem bestimmten Abstand l zum vorhergehenden eingefügt werden muss. Dieser Abstand ist die gewünschte Zeilenlänge. Die Zeilenlänge muss aber nicht exakt eingehalten werden, denn wir vermuten, dass dies unsere Möglichkeiten, überhaupt eine Lösung zu finden, zu sehr einschränken würde. Wir erlauben daher eine gewisse Abweichung d von l und definieren damit einen Bereich, in dem der jeweils nächste Zeilenumbruch eingefügt werden darf. Diesen Bereich nennen wir *Zeilenumbruchfenster*. Die Grenzen des Zeilenumbruchfensters sind die *minimale* und *maximale Zeilenlänge* ($l-d$ und $l+d$). Mit der Bedingung, dass Umbrüche nur im Zeilenumbruchfenster eingefügt werden dürfen, wird der Suchraum eingegrenzt. Die Werte für Zeilenlänge l und erlaubte Abweichung d können fest vorgegeben oder vom Suchverfahren selbst bestimmt werden, im letzteren Fall können sie als Operatoren betrachtet werden.

3.4.3 Absatzoperator

Wie im vorigen Abschnitt beschrieben, soll ein Zeilenumbruchoperator nur im Zeilenumbruchfenster angewendet werden, um die Zeilenlängen möglichst einheitlich zu halten. Es gibt aber einen Fall, in dem ein Text auch mitten in einer Zeile umgebrochen werden kann, dann nämlich, wenn ein neuer Absatz beginnt. Die Schwierigkeit besteht darin, das Ende eines Absatzes zu erkennen. Dafür ist eine semantische Analyse des Textes notwendig und bisher existieren keine Methoden, um solche Stellen zweifelsfrei zu erkennen. Wir vereinfachen daher die Bedingung und erlauben einen Absatzumbruch zwischen Sätzen. Zur Erkennung von Satzgrenzen in einem Text (eng. *sentence decomposition* bzw. *sentence boundary detection* [MS99]) existieren sehr gute statistische Methoden, die wir verwenden, um Satzanfänge und -enden zu markieren.

Ein Absatzoperator kann an jeder Satzgrenze innerhalb der Zeile angewendet werden. Wir notieren einen Absatzoperator (engl. *paragraph*) vor dem i-ten Wort eines Textes mit $\textsc{Pa}(i)$, wie beim Zeilenumbruchoperator reicht auch hier die Positionsinformation aus um den Operator eindeutig zu notieren.

Ein Suchsystem, das nur Zeilenumbruch- und Absatzoperatoren verwendet, sollte in der Lage sein, aus dem unformatierten Text des Schwarzenegger-Briefs (vgl. Abbildung 1.1 auf Seite 5) den korrekt formatierten Text mit dem Akrostichon zu erzeugen. Die Lösung lautet in unserer Notation [$\textsc{Lb}(17)$, $\textsc{Lb}(30)$, $\textsc{Lb}(45)$, $\textsc{Pa}(51)$, $\textsc{Lb}(64)$, $\textsc{Lb}(78)$]. Wir verwenden dies als einfachen Testfall bei der Entwicklung unseres Suchsystems.

3.4.4 Silbentrennoperator

Zeilenumbruch- und Absatzoperatoren können nur die Buchstaben für ein Akrostichon erzeugen, die am Beginn von Wörtern stehen. Betrachten wir erneut das „Hello world“-Beispiel aus Abschnitt 3.4.1. Diesmal wollen wir statt „Hello“ das Wort „Hallo“ als Akrostichon erzeugen. Nur mit Umbrüchen zwischen Wörtern ist dieses Problem nicht lösbar, weil so kein „a“ als zweiter Buchstabe erzeugt werden kann. Es gibt aber ein in Frage kommendes „a“ in dem Wort „example“, denn dieses Wort kann an zwei Stellen getrennt werden: „ex-am-ple“. Ein Silbentrennoperator darf einen Bindestrich und einen Zeilenumbruch zwischen jeweils zwei Silben eines Wortes einfügen und macht die beiden Buchstaben nach den Trennstellen für die Erstellung des Akrostichons verfügbar. Damit können wir eine Lösung finden:

```
Hello world is an easy ex-
ample program to
learn a new programming
language before writing
other, more complex programs.
```

Auch Silbentrennoperatoren sollen nur im Zeilenumbruchfenster eingesetzt werden, denn die Trennung vor dem Zeilenumbruchfenster würde eine zu kurze Zeile erzeugen. Für die eindeutige Notation eines Silbentrennoperators müssen wir nicht nur das Wort sondern auch die Trennstelle im Wort codieren. Wir notieren eine Silbentrennung (engl. *hyphenation*) auf dem i-ten Wort eines Textes an der j-ten Trennstelle als $\textsc{Hy}(i,j)$. Im Beispiel stehen für das Wort „example“ zwei Silbentrennoperatoren zur Verfügung: $\textsc{Hy}(6,1)$ und $\textsc{Hy}(6,2)$. Die Lösung für „Hallo“ wird durch die Sequenz [$\textsc{Hy}(6,1)$, $\textsc{Lb}(9)$, $\textsc{Lb}(13)$, $\textsc{Lb}(16)$] codiert.

Silbentrennoperatoren arbeitet auf Wörtern und beeinflussen nur die Formatierung des Textes. Ihre Leistungsfähigkeit hängt von der Größe des Zeilenumbruchfensters ab, je mehr Wörter in diesen Bereich fallen, desto mehr potenzielle Silbentrennpositionen stehen zur Verfügung. Zur Berechnung der Silben eines Wortes verwenden wir den TEX-Silbentrennalgorithmus [Knu91].

3.4.5 Falschtrennoperator

Die bisher genannten Operatoren (Zeilentrennung, Absatz und Silbentrennung) ändern nichts an der Rechtschreibung der Wörter im Text. Wir haben allgemein den Anspruch, dass die veränderten Texte auch keine Rechtschreibfehler enthalten. Aus gutem Grund haben wir diesen Anspruch nicht als Bedingung formuliert. Wenn wir mit korrekten Operatoren im Suchprozess nicht vorankommen, ist es möglicherweise besser, kleine Fehler zuzulassen, als keine Lösung zu finden.

Wir formulieren einen Falschtrennoperator, indem wir Fehler bei der Silbentrennung zulassen. Ein Falschtrennoperator darf ein Wort an jeder Stelle trennen. Unter der Annahme, dass es weniger auffällig ist, wenn mindestens zwei Buchstaben am Beginn und Ende eines Wortes nicht getrennt werden, schränken wir die Falschtrennoperatoren auf alle Stellen zwischen dem zweiten und vorletzten Buchstaben eines Wortes ein. Damit werden auch erst Wörter ab einer Länge von vier Buchstaben falsch getrennt. Außerdem sollen Falschtrennoperatoren die Silbentrennoperatoren nur ergänzen und keine gleichen Trennungen erzeugen, die zu redundanten Zuständen führen würden. Betrachten wir als Beispiel das Wort „hyphenation“. Die korrekte Trennung ist „hy-phen-ation“, die falschen Trennstellen entsprechend „hyp-h-e-na-t-i-on“. Für dieses Wort stehen also sechs Falschtrennoperatoren zur Verfügung. Aus den gleichen Gründen wie bei Silbentrennoperatoren sollen auch Falschtrennoperatoren nur im Zeilenumbruchfenster angewendet werden. Wir notieren einen Falschtrennoperator auf dem i-ten Wort eines Textes mit Trennung an der j-ten Stelle analog zum Silbentrennoperator mit $\textsc{He}(i,j)$ (für „hyphenation error“).

3.4.6 Einschätzung der Trennoperatoren

Betrachten wir die Eigenschaften der bisher genannten Operatoren. Mit den Trennoperatoren beeinflussen wir nur die Formatierung des Textes, um ein Akrostichon zu erzeugen. Weder die Grammatik, noch der Sinn des Textes wird beeinflusst. Bis auf den Falschtrennoperator werden auch keine Rechtschreibfehler eingefügt. Dies sind gute Eigenschaften. Wann immer es möglich ist, durch einen dieser Operatoren einen Fortschritt im Suchprozess zu erzielen, sollte dieser (mit Ausnahme des Falschtrennoperators) allen weiteren, noch zu diskutierenden Operatoren vorgezogen werden.

Die Distanz, in der ein Zeilenumbruch von der idealen Zeilenlänge l entfernt eingefügt wird, kann zum Vergleich verschiedener Trennoperatoren herangezogen werden. Je geringer der Abstand ist, desto besser ist der Operator hinsichtlich des Formatierungskriteriums.

Die Stärke der Trennoperatoren hängt davon ab, wie viele Buchstaben eines Textes sie im Durchschnitt „verfügbar“ machen können. Dies ist zunächst allgemein (außer beim Absatzoperator) von der Größe des Zeilenumbruchfensters abhängig, die aber für alle Trenoperatoren gleich ist. Wir können damit Vermutungen über eine Rangfolge der Stärke anstellen. Der Absatzoperator kann nur auf die Buchstaben an Satzanfängen zugreifen und ist damit am schwächsten. Der Zeilentrennoperator arbeitet mit den Buchstaben am Wortanfang, das sind sicher mehr als der Absatzoperator. Der Unterschied zwischen Zeilen- und Silbentrennoperator ist schwer einzuschätzen, denn der Silbentrennoperator ar-

beitet nur mit den Buchstaben am Silbenanfang korrekt getrennter Wörter, aber nicht mit dem Anfangsbuchstaben der ersten Silbe (also dem Wortanfangsbuchstaben). Ein Großteil der häufigsten Wörter der englischen Sprache[1] haben zudem keine korrekte Silbentrennung. Wahrscheinlich ist der Silbentrennoperator ähnlich stark wie der Zeilentrennoperator, eine klare Aussage setzt aber umfassende statistischen Analysen voraus. Für den Falschtrennoperator können wir annehmen, dass er hinsichtlich der verfügbaren Buchstaben am stärksten ist, denn er macht für jedes Wort ab vier Buchstaben Länge mindestens einen Buchstaben verfügbar. Diese Stärke kommt aber auch mit einem höheren Preis. Wenn eine Kombination anderer Operatoren, beispielsweise das Ersetzen eines Wortes und eine anschließende korrekte Trennung ebenso zum Ziel führt, müssen wir abwägen, ob die zwei Operatoranwendungen ohne Fehler (angenommen, die Wortersetzung erzeugt keine Fehler) der einen mit Fehler vorzuziehen sind.

3.5 Konstruktive Suchstrategie

Eine Suchstrategie, die nur Trennoperatoren verwendet, kann das Akrostichon Zeile für Zeile (mit jeder Zeile entsteht ein neuer Buchstabe für das Akrostichon) konstruieren und Backtracking benutzen, wenn die Suche in eine Sackgasse führt. Diese Vorgehensweise ist sehr ähnlich zu dem 8-Damen Beispiel aus Abschnitt 2.3.2. Wir bezeichnen diesen Ansatz als *konstruktive Strategie*. Wenn wir nur Trennoperatoren zur Verfügung haben, können wir aber nur dann Akrosticha erzeugen, wenn die benötigten Buchstaben schon in der richtigen Reihenfolge im Ausgangstext vorhanden und von mindestens einem der Operatoren erzeugbar sind. Wir gehen davon aus, dass dies nur selten der Fall ist. Daher benötigen wir weitere Operatoren, die mehr Buchstaben erzeugen können. Wir wollen die konstruktive Strategie weiter verfolgen und stellen die Bedingung, dass alle Operatoren nur in der *aktuellen Zeile* angewendet werden, das heißt zwischen der letzten Zeilenumbruchposition und der maximalen Zeilenlänge. Mit der konstruktiven Strategie verringern wir die Anzahl der Knoten im Suchraum. Wenn ein beliebiger Text länger als die maximale Zeilenlänge ist, so wird beispielsweise kein Pfad untersucht, auf dem im ersten Schritt das letzte Wort verändert wird. Sollte eine Veränderung des letzten Wortes zur Lösung notwendig sein, so wird ein Pfad existieren auf dem dieses Wort einmal in der aktuellen Zeile liegt. Wir schließen also keine Lösungen aus, nur potenzielle Sackgassen. Formell gesehen verringern wir den Verzweigungsgrad (vgl. Abschnitt 2.4) des Zustandsraumgraphen.

[1] Quelle: `http://en.wikipedia.org/wiki/Common_words`, letzter Abruf 22.8.2012

3.6 Kontextabhängiges Einfügen und Ersetzen

Dieser Abschnitt beschreibt Operatoren, die einzelne Wörter im Kontext der umgebenden Wörter im Text verändern, das heißt, diese Operatoren arbeiten im Allgemeinen auf Phrasen. Warum die Kontextinformation oft notwendig ist, wollen wir am Beispiel der Ersetzung von Wörtern durch Synonyme, also Wörter mit gleicher Bedeutung, betrachten.

Synonyme zu verwenden ist eine naheliegende Möglichkeit, Wörter in einem Text durch andere Wörter zu ersetzen. Wenn wir in unserem „Hello world"-Satz das Wort „world" ersetzen wollen, können wir beispielsweise „earth" oder „planet" verwenden um zumindest annähernd die Bedeutung von „world" zu erhalten (wir wollen hier aus Demostrationszwecken auf die sehr spezielle Bedeutung im Zusammenhang mit einem „Hello world"-Programm verzichten).

Um Synonyme zu finden, können wir Synonymwörterbücher, beispielsweise Merriam-Webster[2] oder WordNet[3] verwenden. Ein solches Wörterbuch liefert für ein Wort w eine Menge an Synonymen s_n, dies lässt sich als Abbildung formalisieren: $w \to \{s_1, \ldots, s_n\}$. Ein spezieller Synonymoperator würde dann ein Element aus der Menge des Synonyme wählen, um w zu ersetzen und damit den Text zu verändern.

Leider ist ein Synonymoperator nicht so einfach umzusetzen, denn ein passendes Synonym zu finden ist für einen Computer eine sehr schwere Aufgabe. Beispielsweise liefern die genannten Wörterbücher als Synonyme für „world" unter anderem {folks, humanity, humankind, public, species, globe, cosmos, earth, creation, macrocosm, nature, universe, existence, reality}. Viele dieser Vorschläge entsprechen nicht unserer oben angedachten Bedeutung von „world". Der Grund dafür ist, dass ein und dasselbe Wort nicht immer die gleiche Bedeutung hat. Um dies zu verdeutlichen betrachten wir einige Phrasen in denen das Wort „Welt" in verschiedenen Bedeutungen auftritt:

- „Die ISS kreist um die Welt.": Gemeint ist die Erde als Himmelskörper.
- „Die ganze Welt schaut die Olympischen Spiele.": Gemeint sind Menschen beziehungsweise die Menschheit.
- „in einer anderen Welt leben": Gemeint ist eine individuelle Realitätserfahrung.
- „die Welt der Insekten": Gemeint ist eine bestimmte Domäne.

In allen Fällen bezeichnet das Wort „Welt" ein anderes Konzept. Einem menschlichen Benutzer der Wörterbücher fällt die Auswahl eines passenden

[2] `http://www.merriam-webster.com/`
[3] `http://wordnet.princeton.edu`

Synonyms leicht, im Allgemeinen werden die verschiedenen Konzepte auch schon kategorisiert. Die Menge von Wörtern, die ein gemeinsames Konzept bezeichnen, wird in WordNet als *Synset* bezeichnet. Die Schwierigkeit bei der maschinellen Verarbeitung besteht darin, das passende Synset auszuwählen. Eine Voraussetzung dafür ist die Identifikation der tatsächlichen Bedeutung eines Wortes. Dieses Problem wird als *word sense disambiguation* bezeichnet und ist bisher nicht zufriedenstellend gelöst. Es gibt verschiedene Ansätze aber keine zuverlässigen Systeme und auch keine einheitliche Auszeichnung, die Methoden sind noch dazu sehr aufwendig und kompliziert [Nav09].

3.6.1 Netspeak

Wir haben für unseren Anwendungsfall ein Verfahren entwickelt, das mit einfachen Mitteln auskommt. Wir nehmen an, dass die Bedeutung eines Wortes mit dem Kontext zusammenhängt, in dem das Wort steht. Die Wortersetzung sollte also den Kontext, das heißt die umliegenden Wörter beachten.

Wir verwenden als Basis für Kontextabhängige Operatoren die an der Bauhaus-Universität Weimar entwickelten Suchmaschine Netspeak[4] [PTS10]. Netspeak implementiert einen effizienten Index für den *Web 1T 5-gram* Korpus[5] von Google [BF06] und integriert auch WordNet um eine Synonymsuche im Kontext kurzer Phrasen zu ermöglichen. Die Phrasenlänge ist dabei auf fünf Wörter begrenzt, da dies die maximale Phrasenlänge im zugrundeliegenden Korpus ist. Darüber hinaus bietet Netspeak nicht nur eine Möglichkeit, nach Synonymen zu suchen, sondern auch kurze Phrasen – bis zu vier Wörtern Länge – durch das Einfügen neuer Wörter zu erweitern. Damit ist Netspeak für unsere Zwecke ein wertvolles Werkzeug. Mehr als vier Wörter Kontext angeben zu können wäre wünschenswert, aber zur Demonstration der Operatoren in unserem System ist Netspeak als Datenquelle ausreichend.

3.6.2 Synonymoperatoren

Um den Kontext mit einzubeziehen, definieren wir Synonymoperatoren nicht als Wortersetzung, sondern als Phrasenersetzung. Wir bezeichnen mit w das Wort, für das ein Synonym gesucht wird. Wir wollen zwischen Kontextwörtern vor und nach w unterscheiden, daher bezeichnen wir die Kontextwörter vor w mit fc und die Kontextwörter nach w mit bc. Eine Phrase ist generell eine geordnete Menge von m führenden Kontextwörtern, dem zu ersetzenden Wort und n folgenden Kontextwörtern: $(fc_m, \ldots, fc_1, w, bc_1, \ldots, bc_n)$. In den

[4] `http://www.netspeak.org/`

[5] Linguistic Data Consortium Katalognummer LDC2006T13

Phrasen der Ergebnismenge bezeichnet w'_j ein Synonym für w. Ein Synonymoperator ist dann die Abbildung einer Phrase auf eine Menge von Phrasen:

$$\begin{aligned}(fc_m, \ldots, fc_1, w, bc_1, \ldots, bc_n) \quad \rightarrow \quad & [(fc_m, \ldots, fc_1, w'_1, bc_1, \ldots, bc_n) \\ & \vdots \\ & (fc_m, \ldots, fc_1, w'_k, bc_1, \ldots, bc_n)]\end{aligned}$$

Wir haben verschiedene Typen von Synonymoperatoren entwickelt, die sich darin unterscheiden, ob als Kontext Wörter vor, hinter oder auf beiden Seiten des zu ersetzenden Wortes w betrachtet werden. Die Typen bezeichnen wir je nach Position des zu ersetzenden Wortes in der Phrase als *Context-Synonym-First*, *Context-Synonym-Last* und *Context-Synonym-Center*. Diese Unterscheidung hat einen wichtigen Grund: Ein Synonymoperator, der Kontextwörter auf beiden Seiten von w betrachtet, kann keine Synonyme für Wörter am Beginn oder Ende eines Textes erzeugen. Speziell am Beginn wollen wir aber in der Lage sein, Wörter zu ersetzen. Wenn der Text beispielsweise mit einem anderen Buchstaben beginnt als das Akrostichon, ist ein Synonym des Wortes am Textbeginn eine Möglichkeit, den richtigen Buchstaben zu erzeugen. Außerdem werden von Netspeak teilweise für das gleiche zu ersetzende Wort abhängig von der Position in der Phrase sowohl verschiedenen als auch unterschiedlich viele Synonyme geliefert. Welche Auswirkungen diese Unterschiede auf die Qualität der Synonyme haben, müssen wir noch genauer untersuchen. Dennoch scheinen uns die Unterschiede ausreichend, um mehrere Synonymoperatortypen zu rechtfertigen.

Wir nehmen an, das der Umfang des Kontexts die Anzahl und Qualität der Synonyme beeinflusst. Dies wird durch Untersuchungen bestätigt [PD05, MH11]. Weniger Kontext führt zu einer höheren Anzahl von Synonymen, aber auch zu einer höheren Wahrscheinlichkeit, dass ein Synonym semantisch falsch ist. Mehr Kontext führt zu weniger, aber mit höherer Wahrscheinlichkeit sinnvollen Synonymen.

Um die Notation nicht zu kompliziert zu machen, haben wir uns entschlossen, nur Context-Synonym-Center-Operatoren mit der gleichen Anzahl an Kontextwörtern auf beiden Seiten zu verwenden. Damit ist nicht nur der Begriff „Center“ eine korrekte Beschreibung für die Position des zu ersetzenden Wortes, es reicht auch ein Wert um den Umfang des Kontexts exakt zu bezeichnen und wir können das gleiche Notationsschema für alle Synonymoperatoren verwenden. Für die Notation müssen wir die Phrase, die ersetzt werden soll, den Umfang des Kontexts und das gewählte Synonym codieren. Die Synonymkandidaten betrachten wir als geordnete Menge (Netspeak liefert die Kandidaten in der Reihenfolge ihrer Häufigkeit), so können wir einen Index angeben.

Die Phrase identifizieren wir über die Position i des zu ersetzenden Wortes w im Text und die Anzahl n der Kontextwörter. Wir schreiben CSYF(i,j,n) für einen Context-Synonym-First-Operator, der auf einer Phrase beginnend mit dem i-ten Wort im Text und n anschließenden Kontextwörtern arbeitet und diese durch das j-te Element der Kandidatenmenge ersetzt. Analog schreiben wir CSYL(i,j,n) für einen Context-Synonym-Last-Operator. Einen Context-Synonym-Center-Operator notieren wir mit CSYC(i,j,n). Dabei steht n für die Kontextwörter vor und nach dem zu ersetzenden Wort, mit $n = 1$ wird beispielsweise ein Wort vor und ein Wort nach w betrachtet.

Ein Beispiel zur Verdeutlichung: Bei der Anfrage „hello #world“ liefert Netspeak [world, earth, man, universe, reality, humans] als mögliche Synonyme für „world“. Das ursprüngliche Wort ist in der Ergebnismenge enthalten, da Netspeak alle an dieser Position passenden Wörter zurückliefert. Die Ergebnisse sind nach Häufigkeit geordnet. Der Operator CSYL(2,1,4) beschreibt die folgende Ersetzung am Anfang unseres Beispieltextes: „Hello world is an easy example“ → „Hello universe is an easy example“.

Die Synonym-Operatoren können überall in der Zeile angewendet werden. Einen direkten Einfluss auf die Buchstaben des Akrostichons haben sie zwar nur im Zeilenumbruchfenster, aber wenn das Synonym länger oder kürzer als das ursprüngliche Wort ist, kann sich das auch indirekt auf die Worte im Zeilenumbruchfenster auswirken.

3.6.3 Einfügeoperatoren

Wie in Abschnitt 3.6.1 schon erwähnt, bietet Netspeak auch die Möglichkeit, Wörter an beliebigen Positionen in Phrasen einzufügen. Die Einfügeposition wird in einer Anfrage an Netspeak durch ein Fragezeichen markiert. Beispielsweise liefert die Anfrage „hello ? world“ die Wörter [cruel, kitty, hello, new, open] als Kandidaten für die markierte Position. Das „hello“ an dritter Position ist kein Fehler, die Phrase „hello hello world“ ist in den Ergebnissen enthalten.

Wir nutzen diese Funktion um Phrasen zu erweitern. Dabei treten ähnliche Probleme mit der Semantik auf, wie sie bereits im Zusammenhang mit Synonymen besprochen wurden (vgl. Abschnitt 3.6). Im Fall des Einfügens können wir nicht entscheiden, ob das neue Wort semantisch in den Satz passt. Netspeak liefert nur statistische Evidenz dass eine Phrase mit dem eingefügten Wort im Kontext der angegebenen Wörter existiert. Zusätzlich kann das neu eingefügte Wort die grammatische Struktur des Satzes beschädigen. Die Einfügeoperatoren sollten also im Vergleich mit den Synonymoperatoren höhere Kosten haben, weil auch das Fehlerpotenzial höher ist.

Die Notation der Operatoren ist ähnlich zu den Synonymoperatoren (vgl.

Abschnitt 3.6.2). Wir betrachten sie ebenfalls als Abbildungen einer Phrase auf eine Menge von Phrasen. Auch hier haben wir verschiedene Typen je nach Position des Kontexts entwickelt. Wir notieren: CPRE(i,j,n) (*Context-Prepend*) für einen Operator, der ein Wort vor der Phrase einfügt, CAPP(i,j,n) (*Context-Append*) für einen Operator, der eine Wort nach einer Phrase einfügt und CINS(i,j,n) (*Context-Insert*) für einen Operator, der ein Wort in der Mitte einer Phrase einfügt. Dabei bezeichnet i eine Position im Text, j den Index des eingefügten Elements aus der geordneten Netspeak-Ergebnisliste und n die Anzahl der Kontextwörter. Für die Position i gilt: Bei der Prepend- und Insert-Variante ist die Einfügeposition vor dem i-ten Wort, bei der Append-Variante nach dem i-ten Wort.

Ein Beispiel zur Verdeutlichung: Netspeak liefert bei der Anfrage „hello ? world“ als Kandidaten [cruel, kitty, hello, new, open]. Der Operator CINS(2,4,1) codiert die Abbildung „Hello world is an easy example“ → „Hello new world is an easy example“.

Die Einfügeoperatoren können in der ganzen Zeile angewendet werden, denn sie haben außer dem direkten Erzeugen neuer Buchstaben für das Akrostichon noch eine indirekte Funktion. Durch das Einfügen verschieben sich die Wörter entlang der Zeile und somit können Wörter in den Einflussbereich der Trennoperatoren geraten, die zuvor für diese nicht erreichbar waren.

3.7 Kontextunabhängige Operatoren

Die zuvor vorgestellten Einfüge- und Synonymoperatoren sind vom Kontext abhängig, damit sind sie komplizierter in der Anwendung und fehleranfällig. Wir betrachten in diesem Abschnitt einige Möglichkeiten, Wörter und Phrasen kontextunabhängig zu ersetzen oder einzufügen, die aber trotzdem nur ein geringes Potenzial haben, Fehler in Semantik oder Grammatik zu erzeugen.

3.7.1 Funktionswortoperator

Die Wörter einer Sprache lassen sich unterscheiden in Inhaltswörter (eng. content words) und Funktionswörter (auch Synsemantika genannt). Die Inhaltswörter transportieren hauptsächlich die Bedeutung des Textes, dazu zählen Substantive, Verben, Adjektive und Adverben. Die Funktionswörter sind dafür zuständig, dass das Satzgefüge grammatisch korrekt ist, dazu zählen Konjunktionen, Partikel, Präpositionen, Modal- und Hilfsverben, Artikel und Pronomen. Die Funktionswörter sind vom Kontext praktisch unabhängig. Wenn wir Gruppen solcher Wörter mit ähnlicher Bedeutung zusammenstellen, erhalten wir eine Datenbasis für eine kontextlose Wortersetzung. Beispiele für solche

Gruppen wären „can, may“, „generally, in general, normally, in theory, mostly“, „so, thus, therefore, consequently, as a result“, „however, nevertheless, yet“ oder „for instance, for example, such as, including“.

Eine Funktionswortersetzung kann als spezielle Form von Synonymen angesehen werden, sie werden allerdings von den verfügbaren Synonymwörterbücher kaum erfasst, da diese sich auf Inhaltswörter konzentrieren. Eine weitere Eigenschaft der Funktionswörter kommt uns zu Gute: Sie werden in der Linguistik als „geschlossene“ Klasse betrachtet, das heißt sie sind in der Entwicklung einer Sprache keinen starken Änderungen unterworfen und ihre Anzahl ist relativ konstant. Außerdem ist die Anzahl überschaubar, eine Sammlung, die zu Forschungszwecken verfügbar ist,[6] verzeichnet etwa 350 Wörter und kurze Phrasen. Diese Sammlung nutzen wir als Ausgangspunkt, um eine Liste zusammengehöriger Funktionswörter (bzw. -phrasen) zu erstellen. Unsere Liste enthält derzeit (Ende August 2012) etwa 40 Gruppen. Diese Liste nutzen wir als Datenbasis für Funktionswortoperatoren.

Die Notation eines Funktionswortoperators folgt dem Schema, das schon zuvor bei Synonymoperatoren und Einfügeoperatoren beschrieben wurde. Wir notieren Fw(i,j,n), dabei ist i die Position des zu ersetzenden Wortes im Text und j der Index des gewählten Elements aus der geordneten Menge der Ersetzungskandidaten. Der Wert n bezeichnet die Länge der Phrase. Dies ist in diesem Fall nicht als Kontext zu interpretieren, wir notieren den Wert um, wie im folgenden Beispiel zu sehen ist, auch kurze Phrasen ersetzen zu können.

Ein Beispiel: Wir haben die Funktionswortgruppe [so, thus, therefore, consequently, as a result] und damit die Möglichkeit „as a result“ durch „so“ zu ersetzen. Der Operator Fw(5,1,3) beschreibt diese Ersetzung in dem folgenden Beispiel: „We use function words, as a result we have new operators.“ → „We use function words, so we have new operators.“

Da die Funktionswortersetzung unabhängig vom inhaltlichen Kontext ist und wir durch die manuelle Erstellung der Liste Kontrolle über die Ersetzungsmöglichkeiten haben, schätzen wir die Fehleranfälligkeit hinsichtlich Grammatik und Semantik geringer ein, als bei allgemeinen Synonymen. Funktionsworte treten außerdem häufig auf, praktisch kein Satz kommt ohne sie aus. Diese Eigenschaften machen die Funktionswortoperatoren sehr stark.

3.7.2 Rechtschreibfehleroperator

Dieser Operator ersetzt ein Wort durch eine fehlerhafte Schreibweise des Wortes. Als Datenbasis nutzen wir eine Liste typischer Rechtschreibfehler wie sie

[6] Quelle: `http://www.sequencepublishing.com/academic.html#function-words`, letzter Abruf 22.8.2012

beispielsweise beim Vertippen auf einer Tastatur entstehen.[7] Die Liste hat 3.225 Einträge. Für manche Worte gibt es mehrere Fehlervarianten, für „maintenance“ beispielsweise „maintenence“ und „maintnance“.

Wir notieren einen Rechtschreibfehleroperator als ERR(i,j) mit i als Position des zu ersetzenden Wortes und j als Index in der Liste der Fehlervarianten.

Die Rechtschreibfehleroperatoren gefährden nur die Rechtschreibung (diese dafür mit Sicherheit) aber weder Grammatik, noch Semantik. Sie erzeugen allerdings auch nur wenige neue Buchstaben für das Akrostichon, da oftmals nur ein oder zwei Buchstaben im ursprünglichen Wort ausgetauscht werden. Sie sind darüber hinaus oft nur im Zusammenspiel mit der Silbentrennung oder der Falschtrennung sinnvoll. Aufgrund dieser Eigenschaften schätzen wir die Rechtschreibfehleroperatoren als schwach ein.

3.7.3 Kontraktion und Expansion

Kontraktion bezeichnet das Zusammenziehen mehrerer Wörter, beispielsweise „am not - ain't, will not - won't, I will - I'll“. Das Erzeugen einer Kontraktion und die Umkehroperation, die wir als *Expansion* bezeichnen, ist eine Form der Ersetzung. Für Kontraktion und Expansion von Wörtern gibt es nur wenige Fälle, die sich mit vertretbarem Aufwand manuell in einer Liste erfassen lassen.

Die Ersetzung ist nicht vom Kontext abhängig und fast nicht fehleranfällig. Die Kontraktion ist immer eindeutig. Leider gilt dies nicht für die Expansion, so kann beispielsweise „he'd“ für „he had“ oder „he would“ stehen. Teilweise lassen sich Expansionen eindeutig spezifizieren, indem eine Phrase angegeben wird, beispielsweise wird „I ain't“ zu „I am not“ aber „we ain't“ zu „we are not“ expandiert. Wir haben Kontraktion und Expansion in zwei Operatortypen aufgeteilt. Unsere Kontraktionsliste hat 140 Einträge, die Expansionsliste hat 100 Einträge.

Einen Kontraktionsoperator notieren wir als CON(i,n). Dabei ist i die Position des ersten Wortes der Phrase im Text und n die Länge der Phrase. Die Ersetzung ist eindeutig, daher muss kein weiterer Wert notiert werden.

Einen Expansionsoperator notieren wir als EXP(i,j,n). Dabei ist i die Position des ersten Wortes der Phrase im Text und n die Länge der Phrase. Die Ersetzungskandidaten (oft ist es nur einer) werden in einer Liste geführt und j indiziert die gewählte Ersetzung.

Ein Beispiel: Wir listen für „I'll“ die Expansionskandidaten [I will, I shall]. Der Operator EXP(1,2,1) bezeichnet die Ersetzung „I'll use contraction.“ → „I shall use contraction.“.

[7] `http://en.wikipedia.org/wiki/Wikipedia:Lists_of_common_misspellings/For_machines`

Die Kontraktionsoperatoren erzeugen gar keine Fehler. Die Expansionsoperatoren sind nur wenig fehleranfällig, nämlich nur dann, wenn mehrere Expansionskandidaten möglich, davon aber einige unpassend sind, wie in dem Beispiel „he'd - he had, he would“. Um hier richtig zu entscheiden, müsste der Sinn des Textes bekannt sein. Dies sind aber nur wenige Fälle, oft ist auch die Expansion korrekt.

Die Kontraktions- und Expansionsoperatoren lassen sich allerdings auch nur selten anwenden, da, verglichen beispielsweise mit Funktionsworten oder Synonymen nur wenige Positionen im Text für die Anwendung zur Verfügung stehen. Wir betrachten sie deshalb auch als schwache Operatoren, schätzen sie aber stärker ein als beispielsweise die Rechtschreibfehleroperatoren. Die Kontraktions- und Expansionsoperatoren können durch den Unterschied in der Buchstabenlänge zwischen kontrahierter und expandierter Form auch einen indirekten Einfluss auf das Akrostichon haben.

3.8 Ideen für weitere Operatoren

In diesem Abschnitt diskutieren wir Operatoren, die noch nicht implementiert wurden. Manche der im Folgenden vorgeschlagenen Operatoren sind ausreichend ausgearbeitet um sie in der nächsten Iteration des Demonstrationssystems zu implementieren und zu erproben, manche existieren bisher nur als Idee.

3.8.1 Satzbeginnmodifikation

Der Anfang von Sätzen kann oft erweitert werden ohne die Semantik stark zu beeinflussen. Wir demonstrieren einige Möglichkeiten an dem Beispielsatz „Hello world is an easy example program to learn a new programming language before writing other, more complex programs.“

Aussagebezug konstruieren Der Satz wird einer Person als Aussage zugeschrieben: „Person said, that Hello world is an easy example ...“. Statt des generischen Person kann an dieser Stelle *I, he, she* oder auch jeder beliebige Name eingesetzt werden, beispielsweise: „Christof said, that Hello world is an easy example ...“.

Zeit-Aussagebezug konstruieren Eine Variante ist, einen Zeit-Aussagebezug zu konstruieren, indem einem Satz „Time Person said, that ...“ vorangestellt wird. Für Time kann beispielsweise „Yesterday“, „The other day“ oder etwa „Last Friday“ eingesetzt werden.

Generischer Satzbeginn Eine weitere Variante ist, einem Satz eine generischen Phrase voranzustellen, beispielsweise „actually“, „normally“, „in general“ oder „in theory“: „Normally Hello world is an easy example ...“, „In general Hello world is an easy example ...“.

Als Datenbasis für diese Operatoren wird nur eine Liste generischer Satzteile mit entsprechenden Variablen benötigt und jeweils eine Liste um die Variablen zu belegen, im Fall der generischen Phrasen werden nicht einmal Variablen benötigt.

Nicht bei jedem Satz wird die Semantik so erhalten wie im Beispiel aber wir schätzen diese Operatoren als sehr stark ein. Mit dem Aussagebezug haben wir die Möglichkeit, an bestimmten Stellen im Text beliebige Buchstaben zu erzeugen. Jeder Textanfang ist auch ein Satzanfang und vor allem am Textanfang sind diese Operatoren extrem wertvoll, da wir hier jeden Anfangsbuchstaben eines Akrostichons fast kostenlos erzeugen können.

3.8.2 Tautologische Füllsätze

In einen Text lassen sich beliebig Sätze einfügen, die einen vorhergehenden Satz bestätigen oder eine allgemeine wahre Aussage machen, beispielsweise „As a matter of fact this is true.“, „He was astonished, when he heard this.“, oder „I didn't know that until now.“

Als Datenbasis für diese Operatoren ist nur eine kleine Liste solcher Sätze notwendig. Angewendet werden können sie prinzipiell nach jedem vorhandenen Satz. Zur Notation reicht die Position i des Wortes im Text, vor dem der neue Satz eingefügt werden soll (das heißt i muss auf das erste Worte eines vorhandenen Satzes zeigen) und der Index j der den einzufügenden Satz in der Liste der Kandidaten bezeichnet.

3.8.3 Spezielle Rechtschreibfehler

Unsere Rechtschreibfehleroperatoren (vgl. Abschnitt 3.7.2) basieren auf einer Liste typischer Rechtschreibfehler. Diese Liste ist zwar recht umfangreich (über 3000 Einträge) aber sie enthält nicht für jedes beliebige Wort einen Eintrag, was die Anwendungsmöglichkeiten einschränkt und sie lässt keine Unterscheidung hinsichtlich der Entstehung der Fehler zu. Um mehr Kontrolle über die Fehler zu bekommen, können wir für verschiedene Fehlertypen verschiedene Operatoren entwickeln.

Ein spezieller Typ von Rechtschreibfehlern sind Tippfehler. Beispielsweise werden auf einer herkömmlichen Tastatur nebeneinanderliegende Tasten verwechselt, dann wird ein „i“ zu einem „u“ oder ein „n“ zu einem „m“. Eine

anderer Fehlertyp sind phonetische Fehler also falsche Schreibweise die aus dem gleichen Klang von Buchstaben resultieren. Typische Beispiele im Deutschen sind „gröhlen“ statt „grölen“ oder „hälst“ statt „hältst“ (Beispiele aus dem Wikipedia-Artikel über Rechtschreibfehler[8]). Ähnlich klingende Buchstabenkombinationen für die Englische Sprache sind beispielsweise „f, ph“, „u, ou“, „ui, oo“ oder „ck, k“.

Für jeden Fehlertyp lässt sich eine Liste mit Buchstabenersetzungen erstellen. Eine solche Liste wiederum kann als Datenbasis für einen Operator dienen, der spezielle Fehler für beliebige Wörter erzeugt.

3.8.4 Abkürzungen und Akronyme

Akronyme und Abkürzungen lassen sich ausschreiben, beispielsweise wird dann „NLP“ zu „Natural language processing“ oder „Prof.“ zu „Professor“ und umgekehrt. Auf der Basis einfacher Listen, die die abgekürzte und ausgeschriebene Variante verzeichnen, können wir also Operatoren erstellen, die Ersetzungen in beide Richtungen ermöglichen.

Die Verwendung von Akronymen ist jedoch nicht so einfach und problemlos, wie es zunächst scheint. Zum einen wäre eine solche Liste sehr umfangreich. Spezielle Suchmaschinen wie beispielsweise AcronymFinder[9] behaupten, über 1 Million Einträge zu verzeichnen. Da Akronyme in aller Regel Namen abkürzen, wird eine solche Liste auch niemals vollständig sein. Zur Auflösung von Akronymen kann man natürlich Anfragen an eine solche Suchmaschine stellen, ähnlich wie wir Netspeak für Synonyme verwenden. Zum anderen ist die ausgeschriebene Form zu einem Akronym selten eindeutig. Das Beispiel „NLP“ kann auch für „Neuro-linguistic programming“ oder „Nonlinear programming“ stehen und ohne genaues Wissen über den Inhalt des Textes lässt sich nicht entscheiden, welche Variante richtig ist.

Im Falle von Wortabkürzungen wie „Prof., Dr., u.a.“ scheint es uns einfacher, eine Liste gebräuchlicher Abkürzungen zu erstellen, da hier nicht so viele Möglichkeiten existieren und die Abbildungen in beide Richtungen eindeutig sind. Dennoch scheint uns der nötige Zeitaufwand zur Erstellung einer solchen Liste verglichen mit dem Nutzen dieser Operatoren bisher zu hoch, denn Akronyme und Abkürzungen beziehungsweise Wörter, die für eine Abkürzung in Frage kommen, treten in normalen Texten nur sehr selten auf. Die Abkürzungsoperatoren wären also auch nur sehr selten anwendbar.

[8] `http://de.wikipedia.org/wiki/Rechtschreibfehler`, letzter Abruf 22.8.2012
[9] `http://www.acronymfinder.com`, letzter Abruf 22.8.2012

3.8.5 Zahlen

Zahlen in Ziffernschreibweise lassen sich ausschreiben, beispielsweise „8 - eight, 12 - twelve“. Diese Abbildung ist in beide Richtungen eindeutig und in den meisten Fällen kein Fehler. Als Datenbasis reicht eine Liste der Zahlen, die man üblicherweise ausschreibt. Hier gibt es keine festen Regeln, im Allgemeinen betrifft das die Zahlen von „null“ bis „twelve“ und die Zehnerschritte von „twenty“ bis „one hundred“. Größere Zahlen auszuschreiben ist unüblich, weil sie dann schlecht lesbar sind. Das ist schade, weil ein Zahl wie „one thousand two hundred thirty four“ viele Buchstaben zur Verfügung stellt. Sollen beliebige Zahlen ausgeschrieben werden, ist eine Liste unpraktisch, diese Aufgabe lässt sich recht einfach automatisieren.

Diese Operatoren schätzen wir aber ähnlich schwach ein wie Abkürzungen da auch Zahlen nur sehr selten vorkommen.

3.8.6 Grammatikoperatoren

Wir haben auch in Betracht gezogen, grammatische Kategorien von Wörtern, wie Person, Zahl, Zeitform, Fall oder Modus zu verändern. Diese Veränderungen werden mit dem Begriff Beugung (bzw. Flexion) zusammengefasst. Allgemein ist dazu jedoch die Kenntnis der Wortart notwendig, denn die Änderung einer bestimmten grammatischen Kategorie betrifft oft eine Wortart in besonderer Weise. Beispielsweise müssen zur Veränderung der Zeitform vor allem Verben geändert werden, während zur Falländerung Präpositionen geändert werden müssen. Zur Ermittlung der Wortarten (engl. part of speech, POS) können wir beispielsweise POS-Tagger einsetzen, die zu den Standardwerkzeugen des NLP gehören. Dann können für bestimmte Änderungen simple Regeln angewendet werden. Beispielsweise wird der Plural bei englischen Substantiven oft durch Anhängen eines „-s“ gebildet („operator, operators“) oder die Vergangenheitsform bei Verben durch Anhängen eines „-ed“ („call - called, walk - walked“). Mit solchen einfachen Regeln ist die Wahrscheinlichkeit, grammatisch falsche Formen zu erzeugen aber recht hoch, deshalb sollten zusätzlich Listen eingesetzt werden. Eine Liste irregulärer Verben mit Beugungsformen findet sich beispielsweise in der Wikipedia[10]. Die Liste enthält etwa 500 Einträge und muss manuell aufbereitet werden, was einen erheblichen zeitlichen Aufwand bedeutet und dieser Aufwand erlaubt nur Zeitformoperatoren. Die weiteren grammatischen Kategorien benötigen ähnlich viel Aufwand, daher haben wir die Entwicklung von Grammatikoperatoren als zukünftige Aufgabe zurückgestellt.

[10] Quelle: `http://en.wikipedia.org/wiki/List_of_English_irregular_verbs`, letzter Abruf 22.8.2012

3.8.7 Typografische Operatoren

Eine Möglichkeit zur Veränderung der Formatierung des Textes, abgesehen von Zeilenumbrüchen, ist der Abstand zwischen Worten beziehungsweise Buchstaben. Im Rahmen dieser Arbeit arbeiten wir nur mit nichtproportionaler Schrift, das heißt alle Zeichen sind gleich breit. Wir könnten hier Operatoren formulieren, um die Anzahl der Leerzeichen zwischen Wörtern zu erhöhen. Diese Operatoren haben zwei Anwendungsfälle: Sie können Wörter in den Zeilenumbruchbereich schieben oder die Zeilenlängen angleichen.

Wenn wir proportionale Schrift betrachten, beziehungsweise den Zeichensatz frei wählen können, ergeben sich mehrere neue Möglichkeiten, die hier in Kürze angesprochen, aber nicht ausführlich diskutiert werden sollen. Wir können beispielsweise für den ganzen Text die Schriftfamilie, die Schriftgröße oder den Schriftschnitt verändern. Im Text besteht die Möglichkeit, die Laufweite, das heißt Buchstabenabstände oder Wortabstände, zu verändern. Alle diese Möglichkeiten können als typografische Operatoren formuliert werden.

3.9 Sprachabhängigkeit

Die meisten vorgestellten Operatoren – alle außer Zeilentrennung und Absatztrennung – sind sprachabhängig. Wir betrachten im Rahmen dieser Arbeit nur englische Texte, daher lassen sich die Operatoren nur auf englische Texte anwenden. Die Ideen sind aber auf andere Sprachen übertragbar. In diesem Abschnitt betrachten wir, welche Maßnahmen nötig sind, um auch Texte anderer Sprachen zu verändern.

In den meisten Fällen muss die Datenbasis der Operatoren angepasst werden. Sehr aufwendig ist dies bei den Synonym- und Einfügeoperatoren. Für die Sprache müsste ein System wie Netspeak zur Verfügung stehen. Netspeak basiert auf einem Korpus von n-Grammen mit angegeben Häufigkeiten. Wenn für eine Sprache ein solcher Korpus vorliegt, lässt sich auch ein angepasstes Netspeak entwickeln, die Technologie ist gut dokumentiert [Tre08]. Ein deutsches Netspeak wird beispielsweise gerade an der Bauhaus-Universität Weimar entwickelt. Die Silben- und Falschtrennoperatoren werden auf Basis des TEX-Silbentrennalgorithmus berechnet, der wiederum mit sprachspezifischen Patterns arbeitet. Diese Patterns sind für eine Reihe von Sprachen verfügbar[11]. Weitere Operatoren, wie beispielsweise die Funktionswortoperatoren, basieren auf Listen, die für eine andere Sprache neu erstellt werden müssen.

[11] Eine Sammlung listet die Website `http://tug.org/tex-hyphen/`, letzter Abruf 22.8.2012.

3.10 Aufwandsabschätzung

Wir wollen die Größe des Suchraumes betrachten, der mit den vorgestellten Operatoren aufgespannt werden kann. Dazu schätzen wir die Anzahl der speziellen Operatoren, die in einem Schritt zur Verfügung stehen. Angenommen wir haben als Startzustand einen Text mit 100 Wörtern und die durchschnittliche Länge eines Wortes beträgt 5 Buchstaben (das entspricht etwa der durchschnittlichen Wortlänge der englischen Sprache von 5,2 Buchstaben). Wir ignorieren zunächst die Zeilen und Zeilenumbruchbereiche. Wir können zwischen je zwei Wörtern einen Zeilenumbruch einfügen, das heißt wir erzeugen 100 Zeilenumbruchoperatoren. Angenommen jedes Wort kann an einer Position korrekt getrennt werden, so erzeugen wir weitere 100 Silbentrennoperatoren. Die Falschtrennung liefert mindestens weitere 100 Operatoren, da bei einer Wortlänge von 5 Buchstaben mindestens eine Falschtrennposition zur Verfügung steht. Die weiteren Operatoren schätzen wir auf Basis einiger Stichproben. Die Synonymoperatoren addieren sich ebenfalls zu etwa 100. Die Anzahl der Einfügeoperatoren ist noch höher, vor allem mit einem Kontext von nur drei Wörtern gibt es teilweise für eine Position über 30 Kandidaten (die leider nur selten semantisch in den gesamten Satz passen). Wir schätzen etwa 250 Einfügeoperatoren, wobei dieser Wert sehr tief angesetzt ist. Die Rechtschreibfehleroperatoren addieren sich zu etwa 100 und die Funktionswortoperatoren zu etwa 50. Damit summieren sich die genannten Operatoren auf 800, das heißt, aus dem Startzustand lassen sich 800 Folgezustände erzeugen. Für jeden Folgezustand stehen noch 799 Operatoren zur Verfügung usw. Die Anzahl der Elemente im Suchraum lässt sich für eine Pfadlänge k durch $\frac{800!}{(800-k)!}$ abschätzen. Mit der Einschränkung der Operatoren auf den Bereich einer Zeile beziehungsweise auf das Zeilenumbruchfenster wird die Anzahl verfügbarer Operatoren verringert, der Wert ist dann abhängig von der Zeilenlänge. Bei einer durchschnittlichen Zeilenlänge von 80 Zeichen stehen beispielsweise mehr als 100 Operatoren in jedem Schritt zur Verfügung. Wir müssen also mit einem sehr hohen Verzweigungsfaktor bei der Suche umgehen können. Zum Vergleich: Der durchschnittliche Verzweigungsfaktor bei einem Zug in einer Schachpartie ist 35. Solche Suchräume lassen sich mit uninformierten Verfahren nicht effizient durchsuchen, daher verwenden wir eine Best-First-Suche.

3.11 Paraphrasierung

Das Umformulieren von Texten beziehungsweise das Ausdrücken eines Sachverhaltes mit anderen Worten, wird als *Paraphrasierung* (engl. paraphrasing) bezeichnet und ist ein Forschungsbereich der Computerlinguistik.

Unsere Aufgabe, einen Text so zu verändern, dass er ein Akrostichon enthält,

kann als Paraphrasierung betrachtet werden. Wir haben daher überlegt, Methoden aus diesem Forschungsgebiet anzuwenden. In diesem Abschnitt wollen wir unsere Gründe diskutieren, stattdessen eigene Methoden zu entwickeln.

Die Methoden zur Paraphrasierung basieren zum Großteil auf vorhandenen Textkollektionen. Dies sind teilweise Kollektionen mit mehreren Sätzen, die den gleichen Sachverhalt behandeln, beispielsweise Nachrichtenartikel zum gleichen Ereignis aus verschiedenen Quellen. Die zusammengehörigen Sätze sind üblicherweise markiert. Solche Kollektionen werden als *Parallel corpora* oder *Comparable corpora* bezeichnet [BM01, BL03, CB08]. Varianten benutzen verschiedene Übersetzungen einer Quelle aus einer anderen Spache [PKM03] oder sprachübergreifende parallele Korpora [BCB05]. Zur Unterstützung der Forschung und Evaluation der Methoden wurden einige parallele Korpora der Forschungsgemeinde zugänglich gemacht, beispielsweise der „METER Corpus“ (MEasuring TExt Reuse) [GFW+01] oder der „Microsoft Research Paraphrase Corpus“ (MSRP) [DB05]. Die auf parallelen Korpora basierenden Methoden haben jedoch den Nachteil, dass die extrahierten Paraphrasen auf die Domäne der Korpora beschränkt sind. Außerdem ist die Konstruktion paralleler Korpora sehr aufwendig, daher sind die bekannten Kollektionen nicht sehr umfangreich. Der METER Korpus enthält beispielsweise 1.717 Texte, die 773 Ereignissen aus den Themenbereichen Rechtsprechung und Showgeschäft zugeordnet sind. Der MSRP Korpus enthält 5801 Satzpaare aus diversen Themengebieten, die nicht genauer aufgeschlüsselt wurden.

Es gibt auch Methoden, die Paraphrasen aus normalen, nicht parallelen, Textkollektionen extrahieren [PD05, MHZ11]. Diese Methoden benötigen jedoch sehr große Kollektionen, um darin Paraphrasen zu identifizieren. Auf kleinen Texten, wie denjenigen, die wir verändern wollen, können diese Methoden nicht arbeiten.

Aus unserer Sicht sind die Korpusbasierten Methoden unbrauchbar, denn wir brauchen Methoden, die auf beliebige Texte mit gerimgem Umfang angewendet werden können. Dennoch wollen wir nicht die Erkenntnisse der Forschung in diesem Gebiet ignorieren.

Metzler et. al. haben ein System zur Parahrasenextraktion aus sehr umfangreichen Korpora namens „Mavuno“ entwickelt [MH11] und im Rahmen dieser Arbeit eine Kollektion von Paraphrasen, die mit Mavuno aus dem *English Gigaword Fourth Edition* Korpus[12] extrahiert wurden, veröffentlicht. Diese Sammlung enthält Paraphrasierungen für 17.870 Verb-Phrasen.

Eine solche Kollektion ließe sich als Basis für einen „Paraphrasierungs-Operator“ verwenden. Wir wollen aber verschiedene Formen der Paraphrasierung (wie Wortersetzung, Worteinfügung oder Wortvertauschung) unterschei-

[12]Linguistic Data Consortium Katalognummer LDC2009T13

den und genauer kontrollieren können, daher haben wir einen anderen Ansatz gewählt. Wir überlegen, die Paraphrasenkollektion zusätzlich zu verwenden. Stichroben zeigen, dass einige Paraphrasen sehr spezifisch sind, allerdings wird für jede Paraphrase die Bewertung des Mavuno-Systems angegeben, die sich möglicherweise als Indikator für die Spezifizität verwenden lässt. Wir lassen die Untersuchung dieser Vermutung für zukünftige Arbeiten offen.

Basierend auf Synonymwörterbüchern wie WordNet[13] [Fel98] gibt es Paraphrasierungs-Methoden, deren Ansatz uns für unsere Aufgabe geeignet erscheint [BG04, KB06]. Die hier formulierte Idee ist, eine Menge von Synonymen für ein bestimmtes Wort aus einem Wörterbuch zu beziehen und aus dieser Menge unter Beachtung des Kontexts geeignete Synonyme zu wählen. Wir haben diese Idee aufgegriffen und damit die Netspeak-basierten Kontextabhängingen Operatoren entwickelt (Abschnitt 3.6.1ff.).

3.12 Zusammenfassung

In diesem Kapitel wurden verschiedene Möglichkeiten zur Textveränderung als Operatoren formuliert. Dies sind zum einen Trennoperatoren (Zeilentrennung, Absatztrennung, Silbentrennung und Falschtrennung), die Zeilen in einem Text erzeugen. Eine neue Zeile erzeugt auch einen neuen Buchstaben für das Akrostichon und basierend auf dieser Überlegung haben wir eine konstruktive Suchstrategie entwickelt. Um weitere Buchstaben erzeugen zu können haben wir Synonymoperatoren und Einfügeoperatoren entwickelt, die kontextabhängig sind und verschiedene Ersetzungsoperatoren, die kontextunabhängig sind (Funktionswortoperator, Rechtschreibfehler, Kontraktion und Expansion). Außerdem haben wir mehrere Vorschläge für weitere Operatoren gemacht.

Wie einleitend schon erwähnt (Abschnitt 3.1), sehen wir diese Liste als ausbaufähig an. Es gibt viele weitere Möglichkeiten, Textmodifikationen durchzuführen und je mehr Operatoren wir einsetzen können, desto höher ist unsere Chance, beliebige Akrosticha zu erzeugen. Dass wir mit den hier besprochenen Operatoren in der Lage sind, Akrosticha zu erzeugen, zeigen unsere Experimente (vgl. Kapitel 6).

Bei der Betrachtung der Operatoren zeigt sich, das prinzipiell jeder Operator einen Einfluss auf fast jede Position im Text haben kann. Die wichtige Ausnahme ist der Anfang des Textes, speziell der erste Buchstabe, denn dieser soll gleichzeitig der erste Buchstabe des Akrostichons sein. Um den ersten Buchstaben zu beeinflussen, stehen nur wenige Operatoren, wie beispielsweise *Context-Synonym-First* (vgl. Abschnitt 3.6.2) und *Context-Prepend* (vgl. Abschnitt 3.6.3) zur Verfügung. Wir vermuten, dass der erste Buchstaben häufig

[13] `http://wordnet.princeton.edu`

darüber entscheidet, ob wir ein Akrostichon erzeugen können, oder nicht. Wir haben spezielle Experimente durchgeführt, um diese Vermutung zu überprüfen. Diese Experimente werden in Abschnitt 6.2 vorgestellt. Da sich diese Vermutung bestätigt, können die in Abschnitt 3.8.1 diskutierten Satzbeginnmodifikationsoperatoren unsere Chance, beliebige Akrosticha zu erstellen, deutlich erhöhen.

Kapitel 4

Akrostichonkonstruktion als Suchproblem

Diese Arbeit behandelt die Konstruktion eines Akrostichons. Der Text, in dem das Akrostichon erstellt werden soll, und das Akrostichon selbst sind dabei frei wählbar. Wir haben diese Aufgabe als Suchproblem formuliert.

In Kapitel 2 haben wir eine Einführung in die Zustandsraumsuche gegeben und die allgemeinen Elemente einer Suchproblemspezifikation vorgestellt: Codierung, Operatoren und Suchstrategie. Unsere Ideen für Operatoren haben wir in Kapitel 3 vorgestellt.

Nun müssen wir effiziente Codierungen für den Text und die Operatoren entwickeln, die in einem Suchverfahren anwendbar sind. Wir achten dabei besonders auf zwei Aspekte: Speicher und Zeitaufwand. Eine speichereffiziente Repräsentation ist notwendig, da wir mit dem begrenzten Speicher unserer Rechner möglichst viele Zustände durchsuchen wollen. Die Repräsentation soll aber auch die effiziente Durchführung der grundlegenden Operationen, wie Knotenexpansion (das Erstellen neuer Zustände auf Basis eines bekannten), Zieltest oder die Berechnung von Heuristiken unterstützen.

Dieses Kapitel beschreibt unsere Codierung und die Implementierung in unserem Testsystem.

4.1 Elemente der Suche

Wir formulieren die Akrostichonkonstruktion als Suchprozess, um die Elemente der Suche zu verdeutlichen. Der Suchraum wird, wie in Kapitel 2 beschrieben, als Zustandsraumgraph betrachtet. Ausgangspunkt der Suche ist der gegebene Text. Dieser Text ist als Startzustand s im Startknoten v referenziert. Durch Anwendung von Operatoren generieren wir Kindknoten v', die neue Zustände s', das heißt modifizierte Texte, referenzieren. Jeder Kindkno-

ten erhält zudem eine Referenz auf seinen Vaterknoten und den verwendeten Operator. Dieser Prozess wird für jeden generierten Knoten wiederholt. Jeder generierte Knoten wird überprüft, ob er die Zielbedingung erfüllt, das heißt, ob der Text das Zielakrostichon enthält. Wurde ein Text mit dem Zielakrostichon generiert, soll er ausgegeben werden. Über die Referenzen auf Vaterknoten und verwendeten Operator wird die vollständige Lösung, das heißt der Pfad vom Startzustand zum Zielzustand nachvollziehbar.

4.1.1 Knotencodierung

Jeder Knoten im Suchraum enthält also Informationen über seinen Vaterknoten, den generierenden Operator und den Zustand. Zusätzlich werden die Pfadkosten im Knoten hinterlegt (vgl. Abschnitt 2.4). Ein Knoten wird damit durch ein Tupel (*Vaterknoten, Operator, Zustand, Pfadkosten*) eindeutig beschrieben.

Betrachten wir eine Codierung für einen Knoten, die wir in der Praxis verwenden können. Zwei der Elemente sind trivial. Jeder Vaterknoten ist ein bereits vorhandenes Knotenobjekt und wird über einen Zeiger referenziert. Die Pfadkosten sind ein einfacher numerischer Wert.

Die Operatoren könnten ebenfalls als Objekte angelegt und über Zeiger referenziert werden. Angesichts der großen Menge an Operatoren, die im Suchprozess generiert werden, haben wir aber einen anderen Weg gewählt. Im Kapitel 3 wurde bereits für die einzelnen Operatortypen aufgezeigt, welche Informationen für eine eindeutige Notation notwendig sind. Wir haben darauf basierend eine Codierung entwickelt, die es uns ermöglicht, einen Operator als numerischen Wert und nur mit dem Speicheraufwand eines Zeigers komplett zu beschreiben. Dieser Code wird in Abschnitt 4.4 im Detail vorgestellt.

Der Zustand wird in einem eigenen Objekt repräsentiert und im Knoten ebenfalls über einen Zeiger referenziert. Die Codierung des Zustandes stellt uns vor eine große Herausforderung, da wir keine Annahmen etwa über die Länge des übergebenen Textes machen können und dementsprechend auch den Speicheraufwand nicht vollständig unter Kontrolle haben. Die Zustandscodierung wird in Abschnitt 4.3 im Detail vorgestellt.

Ein Knoten wird damit durch zwei Zeiger und zwei numerische Werte komplett beschrieben. Diese Repräsentation ermöglicht für eine beliebige Implementierung eine Abschätzung des Speicheraufwandes und darauf basierend eine Abschätzung der maximalen Suchraumgröße, die mit dem verfügbaren Speicher einer Maschine durchsuchbar ist.

Betrachten wir Werte aus der Praxis. Auf aktuellen Maschinen ist ein Zeiger 64 Bit groß. Für Operator und Pfadkosten sind in unserer Beispielimplementierung jeweils 32 Bit reserviert. Ein Knoten wird daher mit 192 Bit, also 24 Byte codiert. Auf herkömmlichen Rechnern mit 8GB RAM könnten wir einen Such-

raum von etwa 350 Millionen Knoten aufbauen, ein Rechner mit 100GB RAM erlaubt etwa 4,4 Milliarden Knoten. Zu beachten ist, dass diese Werte nicht den zusätzlichen Speicheraufwand einberechnen, der für die Repräsentation des Zustandes, das heißt des Textes an sich benötigt wird.

4.1.2 Startknotencodierung

Ein spezieller Knoten ist der Startknoten. Er hat keinen Vaterknoten und keinen generierenden Operator. Wir verwenden das Symbol NONE um dies zu bezeichnen. Der Startknoten referenziert den Startzustand s, das heißt den unmodifizierten Text und hat die Pfadkosten 0. Die Darstellung des Startknotens als Tupel ist $(\text{NONE}, \text{NONE}, s, 0)$.

4.1.3 Pfadkostenberechnung

Die Pfadkosten werden als Summe der Operatorkosten entlang eines Pfades berechnet. Wir legen fest, dass die Operatorkosten immer größer oder gleich 0 sind. Ihr genauer Wert ist abhängig vom Operator. In Kapitel 3 haben wir einige Überlegungen zu den Kosten angestellt. Wir betrachten diese Werte aber als Parameter für die Suche, die in zukünftigen Experimenten erforscht werden müssen. In den ersten Testreihen verwenden wir einheitliche Schrittkosten von 1 für alle Operatoren, das heißt die Pfadkosten eines Knotens entsprechen seiner Tiefe. Diese Wahl der Kosten ermöglicht uns den Einsatz einer einfachen aber konsistenten und effektiven Heuristik (siehe Abschnitt 4.8).

4.2 Text und Akrostichon

Ein Text ist eine Sequenz von Zeichen, genauer gesagt alle in englischer Schriftsprache üblichen darstellbaren Zeichen. Die Repräsentation eines Textes wird notwendigerweise auch einige Sonderzeichen wie etwa einen Zeilenumbruch enthalten. Eine Textzeile ist eine Sequenz von Zeichen, die von einem Zeilenumbruch abgeschlossen wird.

Ein Akrostichon ist ebenfalls eine Sequenz von Zeichen, die am Beginn der Zeilen eines mehrfach umgebrochenen Textes von oben nach unten zu lesen sind. Wir wollen uns für das Akrostichon auf Buchstaben und das Leerzeichen beschränken. Diese Beschränkung hat praktische Gründe, denn es ist unüblich, Zeilen im Text mit etwa Satzzeichen beginnen zu lassen. Man kann diskutieren, verschiedene Zeichen, wie etwa Ziffern, das Apostroph oder öffnende Klammern zuzulassen, wir haben uns aus Gründen der Einfachheit dagegen entschieden. Ebenso unterscheiden wir nicht zwischen Groß- und Kleinschreibung. Wir betrachten die Zeichensequenz „Hello World“ als gültiges Akrostichon, während

„Hello, World“ oder „Hello World!“ mit dem Komma und dem Ausrufezeichen ungültige Zeichen enthalten. Akrosticha mit ungültigen Zeichen werden vom System nicht akzeptiert.

Wir betrachten jeden Text so, dass er ein Akrostichon enthält, nämlich die Sequenz der Buchstaben, die sich im aktuellen Zustand am Beginn der Zeilen befinden. Wir bezeichnen dieses als das *aktuelle Akrostichon*. Wir nehmen an, dass das aktuelle Akrostichon in den seltensten Fällen schon eine sinnvolle Nachricht (etwa ein Wort) bildet. Noch seltener wird es die spezielle Nachricht sein, die wir als *Zielakrostichon* vorgeben. Im Zieltest werden das aktuelle und das Zielakrostichon verglichen und bei Übereinstimmung wird der Zustand als Zielzustand akzeptiert. Bei diesem Vergleich ignorieren wir Groß- und Kleinschreibung. In unserem Beispielsystem werden auch Leerzeichen ignoriert, da wir bisher nicht festgelegt haben, wie Leerzeichen im Akrostichon abgebildet werden sollen. Wir formulieren jedoch in Abschnitt 5.2 den Ansatz, für Akrosticha, die aus mehreren Wörtern bestehen, eine verteilte Suche einzusetzen und dabei die Leerzeichen durch Absätze darzustellen.

4.3 Zustandsrepräsentation

Die Zustände in unserer Anwendung sind Variationen eines Textes. Diese Texte müssen effizient repräsentiert werden. Betrachten wir die Anforderungen, die wir an die Repräsentation stellen, im Detail: Die Zustände müssen vergleichbar sein, so dass Pfade, die zu gleichen Zuständen führen, erkennbar sind. Der Vergleich von Zuständen soll möglichst günstig, das heißt mit geringem Rechenaufwand durchführbar sein, da er im Suchprozess sehr häufig benötigt wird. Auch der Zieltest soll günstig sein. Die Repräsentation soll die Berechnung von Heuristiken unterstützen und die Operatoren sollen möglichst effizient mit der Repräsentation arbeiten können, damit die Knotenexpansion nicht zu aufwendig wird. Außerdem sollen die Zustände möglichst speichereffizient codiert werden.

Eine einfache Repräsentation für Texte, wie etwa eine Zeichenkette, erfüllt diese Anforderungen nur zu geringen Teilen. Die Vergleichbarkeit ist zwar gewährleistet, der Speicheraufwand ist aber sehr hoch, da jedes Zeichen in jedem Zustand gespeichert werden muss. Die Operatoren, die auf bestimmten Zeichen, Wörtern oder Phrasen arbeiten, müssen diese erst in der Zeichenkette identifizieren. Unser erster Ansatz zu einer besseren Codierung ist daher, den Text in seine einzelnen Wörter zu zerlegen und als Liste von Wörtern zu repräsentieren. Diese Idee haben wir zu einer wörterbuchbasierten Textrepräsentation weiterentwickelt, die im folgenden beschrieben wird.

Der Text wird tokenisiert, das heißt in eine Sequenz von Worten und Zei-

chen aufgeteilt, die dann als *Token* bezeichnet werden. In unserem Fall werden alle Satzzeichen, wie Punkt, Komma, Klammern oder Anführungszeichen als einzelne Tokens beibehalten. Für die Satzzeichen führen wir spezielle Symbole ein, etwa COMMA für ein Komma oder DOT für einen Punkt. Leerzeichen (genauer Leerzeichen, Tabulator und Zeilenumbrüche) werden verworfen. Die Tokens werden in ein Wörterbuch eingefügt, und jedem Eintrag eine eindeutige ID in Form eines Zahlenwertes zugewiesen. Ein Wort, das in verschiedenen Schreibweisen auftritt (groß oder klein geschrieben, unterschiedliche grammatische Formen) bekommt mehrere Einträge im Wörterbuch und damit verschiedene ID's. Tritt eine Schreibweise mehrmals exakt gleich auf, wird hingegen kein neuer Eintrag im Wörterbuch angelegt. Der Text wird dann als Liste der ID's repräsentiert.

Ein Beispiel: Der Text „*Hello, world. Hello, again.*“ wird tokenisiert zu [*Hello*, COMMA, *world*, DOT, *Hello*, COMMA, *again*, DOT], das Wörterbuch hat die Einträge $\{0 : \textit{Hello}, 1 : \text{COMMA}, 2 : \textit{world}, 3 : \text{DOT}, 4 : \textit{again}\}$ und der Text wird repräsentiert als Sequenz $[0, 1, 2, 0, 1, 4]$.

Eine kurze Erläuterung zum Begriff *Token*. Die Frage, wie viele Wörter der Text „hello world hello again“ enthält, lässt zwei korrekte Antworten zu, nämlich vier, wenn wir das Auftreten einzelner Wörter zählen, aber nur drei, wenn wir die verschiedenen Wörter zählen.[1] Der Begriff *Token* bezeichnet das Auftreten eines Wortes in einem Text. Der Text besteht also aus vier Tokens. Die unterschiedlichen Wörter werden hingegen als *Typ* bezeichnet. Der Beispieltext besteht aus drei Typen, da ein Wort doppelt auftritt. In dem Wörterbuch gibt es also einen Eintrag für jeden Typ mit einer zugeordneten ID. Die Elemente der Sequenz, die den Text repräsentiert, sind Tokens. Im Rahmen dieser Arbeit verwenden wir allgemein den Begriff Token, um ein Element der Sequenz zu beschreiben. Bei der Erwähnung der ID wird nicht explizit darauf hingewiesen, dass die ID des zugehörigen Typen gemeint ist. Außerdem repräsentieren die ID's nicht nur Wörter, sondern auch weitere Elemente, wie etwa Satzzeichen oder Silben, die bereits im Vorverarbeitungsprozess (siehe Abschnitt 4.5) erzeugt werden. Auch diese Elemente bezeichnen wir als Tokens.

Zusätzlich zu den Elementen, die beim Tokenisieren eines Textes entstehen, enthält das Wörterbuch Einträge und somit ID's für Satzbeginn und Satzende. Die explizite Anfangs- und Endmarkierung von Sätzen wird für die Operatoren benötigt, die nur am Satzbeginn oder -ende angewendet werden können oder ganze Sätze modifizieren.

Weiterhin müssen Zeilenumbrüche vermerkt werden, weil sie die Zeichen markieren, die das Akrostichon bilden. Jeder Buchstabe, der nach einem Zei-

[1] Zur Verwendung der Begriffe Token und Type in der Comuterlinguistik vgl. [MS99].

lenumbruch steht, gehört zum Akrostichon. Die Zeilenumbrüche sind jedoch nicht in der ID-Sequenz aufgeführt, stattdessen wurde eine separate Sequenz der Indizes der ID's vorgesehen, die nach Zeilenumbrüchen stehen. Dies erlaubt einen schnelleren Zugriff auf die Elemente des Akrostichons und beschleunigt somit den Zieltest und Heuristiken, die das aktuelle Akrostichon betrachten. Außerdem lassen sich Operatoren, die nur Zeilenumbrüche modifizieren, effizienter implementieren, da diese nicht die ID-Sequenz modifizieren müssen.

Vergleichen wir die beschriebene wörterbuchbasierte Repräsentation mit einer einfacheren Repräsentation als Sequenz von Zeichen oder Wörtern. Jede Änderung im Text hat auch eine Änderung in der ID-Sequenz zur Folge, während zwei identische Texte auch durch identische ID-Sequenzen repräsentiert werden. Damit ist die Vergleichbarkeit von Zuständen weiterhin gewährleistet. Zum Vergleich zweier Zustände müssen in beiden Fällen im Worst-Case (bei identischen Zuständen) alle Elemente der Liste verglichen werden. Der Aufwand ist also in beiden Fällen linear. Die ID-Sequenz hat jedoch generell weniger Elemente als eine entsprechende Zeichenkette, der Vergleichsaufwand wird also reduziert. Der benötigte Speicher wird ebenfalls reduziert, weil nur noch ein Zahlenwert statt einer Menge von Zeichen für ein Token gespeichert wird. Außerdem enthält die Sequenz keine Einträge für Whitespaces, sonst wäre fast jeder zweite Eintrag die ID des Leerzeichens. Der genaue Wert der Speichereinsparung ist davon abhängig, wie viel Speicher für eine ID und für Buchstaben in der Implementierung verwendet wird. In unserem Beispielsystem, das in Java implementiert ist, wird ein Zeichen mit 16 Bit codiert (UTF-16). Wir benutzen auch einen 16-Bit-Wert für die ID. Unter der Annahme, dass ein Wort etwa 5 Zeichen lang ist (die durchschnittliche Wortlänge eines englischen Textes ist 5,2 Zeichen) und ein Leerzeichen nach jedem Wort folgt, braucht ein Text mit 100 Wörtern (etwa sieben Zeilen Text bei einer durchschnittlichen Zeilenlänge von 80 Zeichen) 1.200 Byte. Die ID-Sequenz repräsentiert den Text in 200 Byte, also in einem Sechstel des Speicheraufwandes.

Die Verwendung von ID's erlaubt es, weitere Informationen mit den Tokens zu assoziieren. Beispielsweise wird mit den ID's assoziiert, ob das Token ein Wort, eine Silbe oder ein Satzzeichen ist, was beim Anwenden verschiedener Operatoren hilfreich ist.

Ein Nachteil dieser Repräsentation ist der höhere Aufwand in der Vorberechnung. Der gegebene Text muss in die ID-Sequenz umgerechnet und alle Operatoren basierend auf den ID's vorbereitet werden. Der Vorverarbeitungsprozess wird in Abschnitt 4.5 detailliert beschrieben. Da die Anzahl der ID's bei einer konkreten Implementierung durch den Zahlenbereich des verwendeten Zahlentyps begrenzt ist, ist darauf zu achten, dass der Bereich ausreichend dimensioniert wird. Dabei müssen nicht nur für die verschiedenen Wörter und

Satzzeichen eines Textes ID's vorgesehen werden, sondern auch für alle Elemente, die bei der Vorberechnung der Operatoren entstehen, das heißt beispielsweise weitere Wörter (für Einfüge- und Synonymoperatoren), fehlerhafte Schreibweisen oder Silben. Die Beispielimplementierung verwendet 16-Bit-Werte, wir können also bis zu 65.536 IDs vergeben.

Trotz dieser Repräsentation bleibt der Speicheraufwand ein Problem der expliziten Zustandsrepräsentation. Die Anzahl der Knoten, die im Laufe eines Suchprozesses erstellt werden, steigt exponentiell, und der meiste Speicher wird für die Repräsentation des Zustands aufgewendet. In Abschnitt 4.7 werden Strategien zur effizienten Knotenverwaltung diskutiert.

4.4 Repräsentation der Operatoren

Die Operatorcodierung muss die Zustandsmodifikation eindeutig beschreiben. Dies ist notwendig, um Pfade im Zustandsraum zu repräsentieren (vergleiche Abschnitt 2.4). Die Operatoren wurden in Kapitel 3 ausführlich diskutiert. Wie dort gezeigt, müssen wir zur Notation eines Operators verschiedene Informationen festhalten.

Als Beispiel betrachten wir die Notation eines Synonymoperators, der das erste Wort einer Phrase durch ein Synonym ersetzt (vgl. Abschnitt 3.6.2): CSYF(i,j,n) (wir nennen diese Operatoren „Context-Synonym-First“). Dabei bezeichnet i die Position des zu ersetzenden Wortes im Text. Der Wert j ist ein Index für die Liste der Synonyme und n bezeichnet die Anzahl der Kontextwörter.

Generell repräsentieren wir einen Operator durch ein Tupel mit vier Elementen: (*Typ, Parameter, Alternative, Index*). Jedes Element wird mit einem Zahlenwert codiert. Die Elemente werden im Folgenden beschrieben.

Typ Der *Typ* gibt an, was für ein Art von Operator codiert wird, im Beispiel ein Context-Synonym-First-Operator. Die Angabe des Operatortyps ist für jeden Operator notwendig.

Parameter Der Parameter wird operatorspezifisch interpretiert. Im Beispiel ist es die Anzahl der Kontextwörter n, im Allgemeinen häufig die Phrasenlänge. In der Beispielimplementierung verwenden wir die Phrasenlängen 3, 4 und 5 für Context-Synonym-First-Operatoren. Nicht alle Operatoren benötigen einen Parameter: Der Zeilenumbruchoperator ist beispielsweise nicht parametrisiert. Ist keine Parameterangabe nötig, so kennzeichnen wir dies mit dem reservierten Parameterwert 0.

Alternative Die Alternative gibt das gewählte Element aus einer Liste von Möglichkeiten an, im Beispiel der Wert j für eines der möglichen Synonyme. Bei Operatoren, die keine Angabe einer Alternative benötigen, verwenden wir auch hier den reservierten Wert 0.

Index Der Index gibt an, welche Position im Text von dem Operator modifiziert wird, im Beispiel der Wert i. In der Betrachtung der Operatoren wurde hier immer die Position eines Wortes im Text (das i-te Wort) referenziert. In unserer Codierung des Textes ist i ein Index in der ID-Sequenz, also eines Tokens (das ist allgemeiner als eine Wortposition, da es auch die ID eines Satzzeichens oder einer Silbe sein kann). Die Angabe des Index ist bei jedem Operator notwendig.

Um die Operatoren effizient zu codieren, haben wir in der Beispielimplementierung den Zahlenbereich der einzelnen Elemente eingegrenzt. Der Operatortyp wird durch ein Byte codiert. Damit können wir bis zu 256 verschiedene Operatortypen codieren. Für den Parameter und die Alternative sind jeweils 4 Bit vorgesehen. Da der Wert 0 reserviert ist, bleiben 15 Werte. Der Index ist mit 16 Bit codiert. Zusammengefasst bilden die Elemente der Operatorcodierung einen 32-Bit-Wert. In der Beispielimplementierung werden sie in einem einzelnen `integer`-Wert gespeichert.

Die Operatorkosten sind nicht im Operator codiert, sondern werden mit dem Operatortyp und dem Parameter assoziiert. Damit haben wir die Möglichkeit, die Kosten unabhängig von der Implementierung anzupassen.

4.5 Vorberechnung

Vor der eigentlichen Suche wird der übergebene Text aufbereitet, das Wörterbuch gefüllt, die Textrepräsentation als ID-Sequenz (vgl. Abschnitt 4.3) erstellt und die Operatoren vorberechnet. Die Vorberechnung der Operatoren dient dabei vor allem der Beschleunigung des Suchprozesses. Der Vorberechnungsprozess wird im Folgenden detailliert beschrieben.

Der Text wird normalisiert, das heißt, sämtliche Zeilenumbrüche, Absätze, Tabulatorzeichen und mehrfache Leerzeichen werden durch ein einzelnes Leerzeichen ersetzt. Das Ergebnis dieses Schrittes ist ein einzeiliger unformatierter Text.

Der normalisierte Text wird in Sätze aufgeteilt. Jeder Satz wird daraufhin tokenisiert, das heißt in eine Liste einzelner Worte und Satzzeichen aufgeteilt. Am Beginn und Ende jeder dieser Listen werden Tokens eingefügt, die Satzbeginn und Satzende symbolisieren. Die Listen werden danach in der Reihenfolge

der ursprünglichen Sätze zusammengefügt. Das Ergebnis dieses Schrittes ist eine Liste von Tokens.

Die Tokenliste wird iteriert und für jedes neue Token ein Eintrag im Wörterbuch angelegt. Dabei wird auch jedem Token eine ID zugewiesen. Gleichzeitig wird eine Liste der ID's erstellt. Die Ergebnisse dieses Schrittes sind ein Wörterbuch und eine Liste von Token-ID's, die genau so lang ist wie die ursprüngliche Tokenliste.

Nach diesem Schritt werden der Startzustand und der Startknoten erstellt. Der Startzustand enthält die Liste der ID's und eine leere Liste von Zeilenumbruchpositionen. Der Startknoten erhält einen Zeiger auf den Startzustand, die Pfadkosten werden auf 0 gesetzt und Vaterknoten sowie Operator auf NONE (vgl. Abschnitt 4.1.2).

Danach werden die Operatoren berechnet. Für die meisten Operatoren wird dabei eine Tabelle angelegt, die die möglichen Abbildungen auflistet. Einige Operatoren, wie beispielweise die Funktionsworte oder Rechtschreibfehler (siehe Abschnitte 3.7.2 und 3.7.1) liegen schon als Tabelle in einer Textdatei vor und werden an dieser Stelle geladen. Bei der Berechnung der Operatoren entstehen neue Tokens, beispielsweise falsch geschriebene Wörter, Synonyme oder Silben. Jedes neue Token wird in das Wörterbuch aufgenommen und bekommt eine ID. Alle Einträge der Operatortabellen bestehen nur aus ID's. Da der Prozess für viele Operatoren ähnlich ist, wird hier beispielhaft die Berechnung der Synonymoperatoren dargestellt.

Als Quelle für Synonyme verwenden wir Netspeak. Die Synonymoperatoren arbeiten auf Phrasen, deren Länge durch einen Parameter n festgelegt werden kann. Zusätzlich unterscheiden wir die Operatoren nach der Position des zu ersetzenden Wortes in der Phrase (vgl. Abschnitt 3.6.1). Wir betrachten hier als Beispiel Operatoren, die das erste Wort einer Phrase ersetzen sollen (wir haben diese Variante als *Context-Synonym-First* bezeichnet). Aus der Liste der ID's werden überlappende n-Gramme erstellt. Jedes n-Gramm wird über das Wörterbuch aufgelöst und mit der Phrase eine Anfrage an Netspeak gestellt. Dabei wird das erste Wort markiert, um für dieses Wort Synonyme zu erhalten. Netspeak liefert eine Liste der Synonyme zurück. Diese Liste wird iteriert, für jedes Synonym ein Eintrag im Wörterbuch angelegt und eine ID generiert. In der Tabelle für *Context-Synonym-First*-Operatoren wird für jedes n-Gramm ein Eintrag angelegt, der das n-Gramm auf die Liste der Synonym-ID's abbildet.

Ein Beispiel: Der Text lautet „hello world, and“. Das Wörterbuch hat dann die Einträge {0 : hello, 1 : world, 2 : COMMA, 3 : and} und der Text wird als ID-Sequenz [0,1,2,3] repräsentiert. Für die Berechnung der Synonymoperatoren werden zwei 3-Gramme erstellt, [0,1,2] und [1,2,3] und über das Wörterbuch zu Phrasen aufgelöst „hello world ,“ und „world , and“. Für „hello“, das erste Wort

in der ersten Phrase, liefert Netspeak als Synonym „hi“. Dieses Wort wird ins Wörterbuch übernommen und bekommt die ID 4. Die Abbildung [0,1,2] → [4] wird in der *Context-Synonym-First*-Operatortabelle eingetragen. Für „world“, das erste Wort in der zweiten Phrase, liefert Netspeak als Synonyme „man, earth, public“. Die Synonyme werden in das Wörterbuch aufgenommen und bekommen die ID's 5, 6 und 7. Die Abbildung [1,2,3] → [5,6,7] wird ebenfalls in der Tabelle eingetragen.

Als letzter Schritt in der Vorberechnung werden die Silbentrenn- und Falschtrennoperatoren berechnet. Dafür wird das Wörterbuch iteriert und alle Wörter in Silben aufgeteilt. Die korrekte Silbentrennung wird mit dem TEX-Silbentrennalgorithmus [Knu91] berechnet, die Falschtrennung basiert auf dem Ergebnis der korrekten Trennung. In die Operatortabellen für Silbentrennung und Falschtrennung werden Abbildungen von einem Wort auf die Sequenz der Silben dieses Wortes eingetragen, jede erzeugte Silbe bekommt dabei auch einen eigenen Wörterbucheintrag.

Ein Beispiel: Das Wörterbuch enthält nur ein Wort, {0 : hyphenation}. Der Silbentrennalgorithmus berechnet die Silben [hy, phen, ation]. Die Silben werden in das Wörterbuch eingetragen und bekommen die ID's 1, 2 und 3. In die Silbentrennoperatortabelle wird die Abbildung 0 → [1,2,3] eingetragen. Zur Berechnung der Falschtrennung werden nur die Stellen betrachtet, die nicht von der korrekten Trennung getrennt wurden, außerdem werden jeweils zwei Buchstaben am Wortbeginn und -ende nicht getrennt. Die Falschtrennung von „hyphenation“ lautet mit diesen Regeln [hyp, h, e, na, t, i, on]. Die „falschen Silben“ werden ins Wörterbuch eingetragen und bekommen die ID's 4 bis 10. In die Falschtrennoperatortabelle wird die Abbildung 0 → [4,5,6,7,8,9,10] eingetragen.

Die Berechnung der Silbentrennoperatoren muss am Schluss der Vorberechnung stehen, damit gewährleistet ist, dass auch alle während der Vorberechnung neu hinzugekommenen Wörter getrennt werden.

Damit ist die Vorberechnung abgeschlossen und die Operatortabellen sind gefüllt. Da dieser Prozess sehr zeitaufwendig sein kann, werden die Tabellen am Schluss der Vorberechnung gespeichert. Dazu berechnen wir direkt nach der Erstellung der Tokenliste einen Hashwert über alle Tokens und ordnen die Tabellen diesem Wert zu. Bei einem erneuten Aufruf des Systems wird der Hashwert über der Tokenliste berechnet und geprüft, ob zu diesem Wert gespeicherte Tabellen vorliegen. Wenn dies der Fall ist, werden die Tabellen geladen statt den gesamten Vorberechnungsprozess erneut zu durchlaufen.

4.6 Operatorenanwendung im Suchprozess

Während der Suche wird jeweils ein Knoten aus OPEN, der Menge der nicht expandierten Knoten, zur Expansion gewählt, um neue Knoten zu generieren. Dieser Prozess läuft in mehreren Schritten ab.

Zuerst wird der Bereich festgelegt, in dem die Änderungen angewendet werden sollen. Der Bereich ist von der Zeilenlänge l und der zulässigen Abweichung d, die die Größe des Zeilenumbruchfensters bestimmt, abhängig. Beide Werte sind als Zeichenanzahl angegeben. In diesem Schritt wird die Anzahl der Zeichen zu Positionen k_n in der Liste der ID's umgerechnet. Da wir mit der Liste der ID's nur Tokengrenzen indexieren können, lassen sich die Zeichenlängen nicht exakt umrechnen. Drei Positionen werden berechnet: der Beginn der Zeile k_0 sowie Beginn k_1 und Ende k_2 des Zeilenumbruchfensters. Der Beginn der Zeile k_0 ist einfach zu ermitteln, da die Position des letzten Zeilenumbruchs in jedem Zustand bekannt ist, denn für diese Positionen wird eine separate Liste geführt. Die Position k_1 soll bei etwa $l - d$ Zeichen und die Position k_2 bei etwa $l + d$ Zeichen liegen. Dazu werden die Zeichenlängen der aufeinanderfolgenden Tokens, beginnend mit dem an Position k_0 aufsummiert. Für jedes Wort wird zusätzlich 1 addiert, um das Leerzeichen nach dem Wort mit einzuberechnen. Die Position des Tokens, bei dem die Summe den Wert $l - d$ überschreitet, wird als k_1 festgehalten. Die Tokenlängen werden weiter aufsummiert, bis die Summe den Wert $l + d$ überschreitet. Die Position des nächsten Tokens wird als k_2 festgehalten. Damit wird das Zeilenumbruchfenster ausgeweitet. Diese Ausweitung ist nötig, um Grenzfälle zu behandeln, in denen ein einzelnes Token durch den gesamten Zeilenumbruchbereich läuft. Dies kann auftreten, wenn ein einzelnes Token länger als $2d$ Zeichen ist. Ein solches Token kann dann beispielsweise mit einem Silbentrennungsoperator modifiziert werden, der nur im Zeilenumbruchfensters angewendet werden kann.

Nachdem die Positionen berechnet sind, werden die verfügbaren Operatoren berechnet. Jede ID an den Positionen zwischen k_0 (beziehungsweise k_1 für die Operatoren, die nur im Zeilenumbruchfenster angewendet werden) und k_2 wird dazu mit den Einträgen in den Operatortabellen verglichen. Für Operatoren, die auf n-Grammen arbeiten, wird vor der Anfrage in der Tabelle der Kontext entsprechend erweitert. Enthält eine Operatorabelle einen passenden Eintrag, so wird der Operator einer Liste anwendbarer Operatoren hinzugefügt. Die Interpretation der Einträge in den Operatortabellen hängt dabei vom Typ des jeweiligen Operators ab. Zur Erläuterung betrachten wir die Beispiele aus Abschnitt 4.5 (die ID's sind hier zum besseren Verständnis aufgelöst). Für einen *Context-Synonym-First*-Operator wird bei der Abbildung [hello, world, COMMA] → [hi] die linke Seite als Phrase interpretiert und die rechte Seite als Liste von Wörtern, die das erste Wort der Phrase ersetzen

können. Hier wird ein spezieller *Context-Synonym-First*-Operator repräsentiert, der die Textstelle „hello world," zu „hi world," verändern kann. Für einen Silbentrennoperator wird bei der Abbildung hyphenation → [hy, phen, ation] die linke Seite als ein trennbares Wort und die rechte Seite als Sequenz der Silben interpretiert. Hier werden zwei spezielle Operatoren repräsentiert, die den Text im Wort „hyphenation" an zwei verschiedenen Stellen umbrechen können, „hy-phenation" und „hyphen-ation".

Im letzten Schritt wird die Liste anwendbarer Operatoren iteriert und jeder Operator angewendet, um jeweils einen neuen Zustand zu erzeugen. Dabei wird üblicherweise eine Sequenz von ID's in der Liste der ID's durch eine andere Sequenz ersetzt. Wird ein Zeilenumbruch eingefügt, so wird die Liste der Zeilenumbruchpositionen erweitert. Jede Liste, die modifiziert wird, wird zuvor kopiert. Wird eine Liste nicht modifiziert, kann sie vom neuen Zustand referenziert werden. Mit jedem neuen Zustand wird ein neuer Knoten generiert und in die Knotenverwaltung eingefügt.

4.7 Knotenverwaltung

Alle Knoten, die im Suchprozess entstehen, müssen während der Suche effizient verwaltet werden. Die grundlegenden Strukturen zur Knotenverwaltung (vgl. Abschnitt 2.4) sind Open (die Menge der generierten Knoten) und Closed (die Menge der expandierten Knoten).

Um eine effiziente Suche zu gewährleisten, müssen beide Datenstrukturen sowohl berechnungs- als auch speichereffiziente Operationen unterstützen. Da wir eine Best-First-Suche einsetzen, ist Open als Prioritätswarteschlange implementiert. Die Knoten werden durch eine Heuristik bewertet und die Struktur von Open garantiert einen effizienten Zugriff auf den am besten bewerteten Knoten. An Closed werden zunächst keine speziellen Anforderungen gestellt.

Beide Mengen wachsen im Laufe der Suche kontinuierlich an und werden irgendwann den gesamten zur Verfügung stehen Speicher vereinnahmen. Daher werden bei einer praktischen Umsetzung Maßnahmen ergriffen, den Speicher möglichst effizient zu nutzen. Wir haben in den Abschnitten 4.3 und 4.4 bereits effiziente Repräsentationen für Zustände und Operatoren vorgestellt. In diesem Abschnitt werden Maßnahmen diskutiert, die wir im Rahmen der Knotenverwaltung einsetzen können.

Wir betrachten die Menge Open der generierten Knoten. Für alle Knoten in Open ist es üblich, den Zustand zu speichern. Trotz unserer Bemühungen um eine effiziente Repräsentation ist der Speicheraufwand für den Zustand deutlich höher als der Aufwand für die anderen Informationen eines Knotens. Wir können also stark davon profitieren, den Zustand nicht in jedem Knoten

explizit zu speichern. Betrachten wir zunächst, wozu wir die Zustandsinformation brauchen:

- Knotenexpansion : Bei der Generierung neuer Knoten muss der Zustand des zu expandierenden Knotens verfügbar sein.
- Zieltest : Der Zustand wird überprüft, ob er das Akrostichon enthält.
- Bewertung : Die Zustandsinformation fließt in die Aufwandsabschätzung des Restproblems ein.
- Zustandsvergleich : Um redundante Pfade zu erkennen wird der Zustand jedes neu generierte Knoten mit allen bekannten Zuständen verglichen, bevor der Knoten in die Knotenverwaltung übernommen wird.

Für drei der vier Punkte gibt es genau einen Zeitpunkt, bei dem der Zustand bekannt sein muss: Jeder Knoten wird nur einmal expandiert, nur einmal dem Zieltest unterzogen und nur einmal bewertet. Zudem liegen diese drei Punkte eng beisammen: Der Zieltest wird entweder für den zur Expansion gewählten (bei einer A*-Suche) oder für einen neu generierten (bei einer generellen Best-First-Suche) Knoten durchgeführt, die Bewertung nur für einen neu generierten. Damit ergeben sich Möglichkeiten zur speichereffizienten Knotenverwaltung, die in den folgenden Abschnitten diskutiert werden.

4.7.1 Zustandsrekonstruktion

Da jeder Knoten einen eindeutigen Pfad vom Startzustand zu dem Zustand des Knotens repräsentiert, kann der Zustand bei der Auswahl eines Knotens zur Expansion rekonstruiert werden. Dann wird der Zieltest durchgeführt, die Kindknoten generiert und bewertet. Die Rekonstruktion eines Zustandes bedeutet zwar einen zusätzlichen Rechenaufwand, dieser ist aber vertretbar, wenn er für jeden Knoten nur einmal durchgeführt wird.

4.7.2 Time-Memory Tradeoff durch Hashwerte

Damit könnte der Zustand verworfen werden, wäre es nicht nötig, den Vergleich mit allen bekannten Knoten durchzuführen. Dieser Punkt stellt das größte Problem in der Knotenverwaltung dar. Wir könnten auch hier alle Zustände rekonstruieren, aber das hieße, für jede Knotenexpansion alle bisherigen Zustände neu zu berechnen. Die Anzahl der Knoten steigt im Suchprozess exponentiell. Daran können wir nichts ändern. Mit dem Verwerfen der Zustandsinformation

verringert sich nur der Exponent im Speicheraufwand, mit der ständigen Rekonstruktion würde aber ein exponentieller Rechenaufwand entstehen. Dieser Aufwand ist nicht vertretbar.

Stattdessen nutzen wir eine andere Strategie, die einen Kompromiss aus Rechen- und Speicheraufwand darstellt. Beim Generieren eines Knotens wird für den Zustand ein Hashwert berechnet, der den Zustand identifiziert. Dieser Hashwert wird zusätzlich im Knoten gespeichert. Damit können wir die Zustandsinformation verwerfen ohne die Vergleichbarkeit zu verlieren. Der Mehraufwand für diese Methode besteht in den Berechnungskosten für den Hashwert, die zu den Kosten für die Knotengenerierung addiert werden sowie den Speicherkosten für den Hashwert, die zu den Speicherkosten für einen Knoten addiert werden. Dies ist vertretbar, da die Speicherkosten für den Zustand entfallen.

Die Verwendung von Hashwerten hat noch weitere Vorteile. Der Aufwand für den Vergleich von Zuständen wird auf den Aufwand für den Vergleich von Hashwerten reduziert. Hashwerte haben eine feste Größe, damit lässt sich der Speicheraufwand für einen Knoten besser kontrollieren als mit den Zuständen, deren Größe variabel ist.

Ein Nachteil bei der Verwendung von Hashwerten ist die Möglichkeit, dass Kollissionen auftreten können, das heißt für verschiedene Zustände wird der gleiche Hashwert berechnet. Tritt eine Kollission auf, bedeutet das in unserem Fall, dass ein Knoten als bereits bekannt verworfen wird, obwohl er einen unbekannten Zustand repräsentiert. Dabei können auch mögliche Lösungspfade verworfen werden. Die Wahrscheinlichkeit für eine Kollission steigt mit der Anzahl der Knoten. Um dieses Risiko zu minimieren sollte eine Hashfunktion verwendet werden, die einen großen Werteumfang hat und die Hashwerte gut verteilt. Mit diesen Eigenschaften sind allerdings auch steigende Kosten zur Berechnung und Speicherung des Hashwertes verbunden. Wir verwenden die Hashfunktion MD5, die 128-Bit-Hashwerte erzeugt. Damit ist das Kollissionsrisiko auf einen praktisch irrelevanten Wert minimiert, der Berechnungsaufwand erhöht jedoch in Pilotexperimenten die Suchdauer um etwa 30% gegenüber einer Suche ohne Hashwertberechnung.

4.7.3 Effizienter Zugriff auf Elemente in der Knotenverwaltung

Eine weitere Anforderung an die Knotenverwaltung im Zusammenhang mit dem Zustandsvergleich ist der effiziente Zugriff auf die bekannten Knoten. Alle Knoten in OPEN und CLOSED zu durchsuchen, ob ein neu generierter Zustand schon bekannt ist, wäre deutlich zu aufwendig, denn die Anzahl der zu durchsuchenden Elemente steigt im Laufe der Suche exponentiell. Einen

effizienten Zugriff bietet eine Hashtabelle. Da OPEN bereits als Prioritätswarteschlange strukturiert ist, ist die Verwaltung als Hashtabelle ein zusätzlicher, aber notwendiger Aufwand. Für CLOSED ist der effiziente Zugriff die einzige Anforderung, die wir stellen, daher ist CLOSED explizit als Hashtabelle strukturiert.

4.7.4 Caching von Zuständen

Die Zustandsrekonstruktion kann sehr rechenaufwendig sein. Um den Suchprozess zu beschleunigen, bietet es sich daher an, einen Teil der Zustände in einem Cache zwischenzuspeichern. Zur Identifikation der Zustände, die im Cache sind, können wir die Zustands-Hashwerte verwenden. Zur Auswahl der Zustände, die in den Cache sollen, lässt sich die in Abschnitt 4.8 beschriebene Heuristik zur Knotenbewertung einsetzen: Die am besten bewerteten Knoten sind Kandidaten für die Expansion, deshalb ist es sinnvoll, die betreffenden Zustände vorzuhalten.

4.8 Suchprozess und Heuristik

Unser generelles Vorgehen ist die Konstruktion des Akrostichons Buchstabe für Buchstabe, daher bezeichnen wir diese Strategie als *konstruktive Suchstrategie* (sie wurde in Abschnitt 3.5 schon angesprochen). Wir gehen von einem unformatierten Text aus, und konstruieren das Akrostichon durch die Anwendung von Trennoperatoren, die den Text am Ende einer Zeile umbrechen. Dabei soll nicht an beliebigen Positionen umgebrochen werden, sondern nur innerhalb eines bestimmten Bereiches um eine bestimmte Zeilenlänge. Zeilenlänge und Umbruchbereich können fest vorgegeben, oder als Parameter vom Suchsystem gewählt werden. Die Operatoren, die keine Umbrüche erzeugen, werden verwendet, um einen Zustand zu erstellen, in dem das Umbrechen einen Fortschritt in der Akrostichonkonstruktion ermöglicht.

Wir verwenden eine Best-First-Suche, wie sie in Abschnitt 2.5.3 vorgestellt wurde. Die wichtigste Komponente ist dabei die Heuristik, die Knoten bewertet und damit bestimmt, welcher Knoten als nächster expandiert wird. Wir haben eine sehr einfache Heuristik entwickelt, die im Folgenden vorgestellt wird.

Die Heuristik f berechnet den Wert eines Knotens v als Summe der Pfadkosten g und der Kostenabschätzung h für das Restproblem: $f(v) = g(v) + h(v)$. Die Pfadkosten g berechnen sich als Summe der Kosten der jeweils verwendeten Operatoren auf dem Pfad vom Startknoten bis v. Im Beispielsystem verwenden wir einheitliche Kosten von 1 für alle Operatoren. Damit sind die Pfadkosten äquivalent zur Länge des Pfades. Für die Restkostenabschätzung

h subtrahieren wir die Anzahl der übereinstimmenden Buchstaben von Zielakrostichon und aktuellem Akrostichon von der Länge des Zielakrostichons. Da unser konstruktives Vorgehen sicherstellt, dass nur Buchstaben am Beginn des aktuellen und Zielakrostichon übereinstimmen, entsprechen die Restkosten also der Anzahl der noch nicht erzeugten Zielakrostichonbuchstaben. Wenn das Zielakrostichon beispielsweise „hello“ lautet und in einem Zustand das aktuelle Akrostichon „he“ erzeugt wurde, schätzen wir h mit 3 ab. Die Knotenauswahlfunktion wählt immer den Knoten mit dem geringsten Wert für f zur Expansion, bei mehreren Knoten mit gleichem f wird zufällig ausgewählt.

Da wir das Akrostichon Buchstabe für Buchstabe konstruieren und als Restkostenabschätzung die noch fehlenden Buchstaben betrachten, bezeichnen wir diese Heuristik als CML („Constructive Missing Letters“).

Betrachten wir die Eigenschaften der CML-Heuristik: Die Pfadkosten sind immer positiv, und die Restkostenabschätzung kann die tatsächlichen Kosten des Restproblems nie überschätzen, denn für jeden zu erzeugenden Buchstaben muss mindestens ein Operator angewendet werden. Damit ist die CML konsistent und die Suche ein A*-Algorithmus (vgl. Abschnitt 2.5.4).

Durch das konstruktive Vorgehen haben wir auch eine einfache Möglichkeit, Sackgassen frühzeitig zu erkennen: Wenn das aktuelle Akrostichon in einem Zustand länger ist als die Anzahl der übereinstimmenden Buchstaben mit dem Zielakrostichon, kann dieser Zustand verworfen werden.

Wenn eine Lösung gefunden wird, so ist garantiert, dass sie auf einem kürzesten möglichen Pfad liegt, das heißt, es wurde möglichst wenig am ursprünglichen Text verändert. Wir nehmen an, dass mit der Anzahl der Änderungen auch die Anzahl der Fehler (in Rechtschreibung, Grammatik oder Semantik, vgl. die Kriterien in Abschnitt 3.2) steigt, die in den Text eingebaut werden. Dann wäre eine Lösung mit möglichst wenigen Änderungen wahrscheinlich auch eine gute Lösung. Diese Annahme ist aber mit Vorsicht zu betrachten, da wir Operatoren verwenden, die mit Sicherheit Fehler erzeugen, aber auch sehr häufig einen Fortschritt im Suchprozess bringen, wie beispielsweise den Falschtrennoperator (vgl. Abschnitt 3.4.5).

Mit dem in diesem Kapitel beschriebenen Suchverfahren können wir einen Machbarkeitsnachweis für die Idee der Akrostichonerzeugung mittels heuristischer Suche erbringen. Wir haben dazu ein Testsystem implementiert, das wir auch als Basis für weitere Entwicklungen betrachten. Wir wollen untersuchen, wie gut wir mit diesem System Akrosticha erzeugen können. Dafür haben wir eine Reihe von Experimenten entworfen, die in Kapitel 6 beschrieben werden. Die Experimente sollen sowohl zur Evaluation des Systems, als auch als Benchmark für zukünftige Verbesserungen dienen.

Verbesserungen sind möglich, denn das oben beschriebene Verfahren hat

auch zwei klare Nachteile, die beide durch die Methode der Berechnung der Pfadkosten entstehen. Zum einen ist die Neigung der einzelnen Operatoren, Fehler zu erzeugen, nicht in den Operatorkosten abgebildet. Damit erzeugen wir als „beste“ Lösung eine, die mit wenig Aufwand konstruierbar ist, aber nicht unbedingt eine, die wenige Fehler enthält. Zum anderen ist der Speicheraufwand sehr hoch, denn alle Knoten müssen erzeugt werden, bevor sie bewertet werden können. Solange kein neuer Buchstabe für das Akrostichon erzeugt wird, entspricht die Suche durch die einheitlichen Pfadkosten aber einer Breitensuche. Wir führen also pro Zeile eine Breitensuche durch und speichern alle erstellten Knoten, das heißt die Heuristik ist sehr schwach. Wir diskutieren mögliche Verbesserungen des Verfahrens im folgenden Kapitel.

Kapitel 5

Bessere Suchverfahren

Mit dem in Kapitel 4 vorgestellten Suchverfahren sind wir in der Lage, Akrosticha zu konstruieren, wie unsere Experimente (Kapitel 6) zeigen. Wir sehen allerdings auch viel Raum für Verbesserungen. In diesem Kapitel betrachten wir die Probleme und diskutieren Möglichkeiten, das Suchverfahren auszubauen und zu verbessern.

5.1 Problemanalyse

Betrachten wir die Probleme, die sich in den Experimenten abzeichnen. Wir können viele Akrosticha nicht konstruieren, das heißt wir finden kein Lösungsobjekt im Suchraum. Das kann zwei Gründe haben: Es existiert kein Lösungsobjekt oder wir haben nicht die Ressourcen, eines zu finden.

Es ist möglich, dass kein Lösungsobjekt existiert, weil die Operatoren keines erzeugen können. Wenn wir beispielsweise nur Trennoperatoren einsetzen, der Text aber einen der für das Akrostichon benötigten Buchstaben gar nicht enthält, lässt sich das Akrostichon nicht konstruieren. Deshalb setzen wir auch Operatoren ein, die neue Worte und somit neue Buchstaben einfügen können, wie die Synonymersetzungs- oder Worteinfügeoperatoren. Diese Operatoren ändern allerdings nichts an der Wahrscheinlichkeit, mit der ein bestimmter Buchstabe im Text vorkommt. Seltene Buchstaben werden daher auch nur mit geringer Wahrscheinlichkeit eingefügt. Seltene Buchstaben treten aber auch im Akrostichon nur mit geringer Wahrscheinlichkeit auf. Wahrscheinlicher ist es, dass ein Buchstabe zwar vorhanden ist, aber nicht an der richtigen Stelle steht. Fast alle Operatoren (mit Ausnahme der Trennoperatoren) sind in der Lage, die Position von Buchstaben im Text zu verändern. Wir haben in Abschnitt 3.8 bereits einige weitere Operatoren vorgestellt, die noch nicht implementiert sind. Weitere Operatoren erhöhen die Größe des Suchraumes und damit die Wahrscheinlichkeit, dass Lösungsobjekte existieren.

Das führt zum zweiten möglichen Grund: Dem Mangel an Ressourcen, das heißt Speicher und Zeit. Bei der Implementierung haben wir darauf geachtet, einen möglichst guten Kompromiss zwischen Speicherverbrauch und Zeitaufwand zu finden. Der Suchraum ist aber dennoch deutlich größer, als wir mit dem zur Verfügung stehenden Speicher erfassen können. Eine grobe Schätzung mit 100 anwendbaren Operatoren pro Zeile ergibt für ein Akrostichon von fünf Buchstaben Länge bereits 100 Millionen Knoten (mindestens vier Zeilenumbrüche müssen erzeugt werden, in der fünften Zeile muss nicht weiter gesucht werden, wenn der richtige Buchstabe am Beginn steht). Bei einem angenommenen Speicherverbrauch von 100 Byte pro Knoten belegt diese Menge 9,31 GB Speicher und damit mehr als handelsübliche Rechner zur Verfügung haben (die Standardausstattung liegt derzeit bei 8 GB). In unseren Experimenten brechen wir die Suche ab, wenn der Speicher voll ist oder eine bestimmte Anzahl Knoten untersucht wurde. Letztere Bedingung ist mit einer zeitlichen Grenze vergleichbar. Um den Herausforderungen begrenzter Ressourcen zu begegnen, müssen wir das Suchverfahren verbessern. Wir sehen hier zwei Ansätze: Verteilung der Suche und Einsatz besserer Heuristiken. Diese Ideen werden in den folgenden Abschnitten diskutiert.

5.2 Verteilte Suche

Die grundlegende Idee der verteilten Suche ist es, den Suchraum in kleinere Abschnitte aufzuteilen und die einzelnen Teile zu durchsuchen. Dies kann parallel oder sequenziell durchgeführt werden, wobei die parallele Suche, also die Aufteilung auf mehrere Maschinen, den Vorteil hat, dass eine Lösung damit auch in kürzerer Zeit gefunden werden kann.

Ein grundlegendes Problem bei der Aufteilung ist, zu verhindern, dass sich die Bereiche überschneiden, und damit die Schnittmenge unnötig mehrfach durchsucht wird. Allgemein wird das durch die Knotenverwaltung verhindert, das heißt alle bekannten Knoten werden gespeichert (in den Mengen Open und Closed, vergleiche Abschnitt 2.4) und bei neu generierten Knoten wird abgeglichen, ob sie schon bekannt sind. Diese Knotenverwaltung führt aber in der Praxis gerade zu den erwähnten Speicherproblemen. Beim sequenziellen Durchsuchen der Abschnitte müssten die bekannten Knoten aus jedem Abschnitt des Suchraumes für alle weiteren Suchvorgänge gespeichert werden, was diese Idee ad absurdum führt. Bei einer parallelen verteilten Suche müsste die Information über bekannte Knoten zwischen allen Maschinen ausgetauscht werden, was sehr aufwendig ist.

Sinnvoll wird eine Aufteilung dann, wenn man in der Lage ist, ein Problem in unabhängige Teilprobleme zu unterteilen, die mit geringerem Aufwand

lösbar sind. Dieser Ansatz wird auch als *Problemreduktion* bezeichnet. Eine Aufteilung in Teilprobleme wird besonders dann nützlich, wenn wir längere Akrosticha, etwa Phrasen oder ganze Sätze erzeugen wollen. Wir betrachten einige Möglichkeiten, wie unser Problem der Akrostichonkonstruktion aufteilbar ist.

Wir benötigen eine Methode, um den Problemaufwand einzuschätzen. Dies wird später im Detail diskutiert. Hier gehen wir vereinfachend davon aus, dass ein Akrostichon schwieriger zu konstruieren ist, je länger es ist. Damit ergibt sich schon ein grundlegender Ansatz zur Aufteilung: Das Akrostichon und der vorgegebene Text werden aufgeteilt, jeweils ein Teil des Akrostichons in einem Teil des Textes konstruiert und die Teillösungen wieder zusammengefügt. Zur Aufteilung des Akrostichons und des Textes haben wir verschiedene Möglichkeiten. Wenn das Akrostichon aus mehreren Wörtern besteht, können wir jedes Wort einzeln konstruieren. Der Text lässt sich in diesem Fall etwa an Satzgrenzen aufteilen, also mit dem Absatzoperator (beschrieben in Abschnitt 3.4.3). Ein Absatz zwischen den Teillösungen hätte den Vorteil, dass die Trennstelle zwischen jeweils zwei Wörtern im Resultat auch optisch dargestellt werden kann, so wie es beispielsweise in dem Schwarzenegger-Brief (Abbildung 1.1 auf Seite 5) der Fall ist. Wie viel Text für jeden Teil notwendig ist, lässt sich über die Zeilenlänge abschätzen. Soll die durchschnittliche Zeile etwa 80 Zeichen lang sein und das Akrostichon hat vier Buchstaben. So sind zwischen 240 (drei volle Zeilen idealer Länge) und 360 Zeichen Text (vier volle Zeilen) eine sinnvolle Wahl. Diese Idee lässt sich weiter verfolgen: Wenn das Akrostichon nur ein Wort ist, kann es in Silben aufgeteilt werden und der Text jeweils in eine Anzahl „Pseudozeilen“, die der Silbenlänge entspricht. Im Extremfall kann jede Zeile und damit jeder Buchstabe des Akrostichons separat betrachtet werden. Wir betrachten diese Ansätze als zukünftige Forschungsfragen. Die Idee der Aufteilung in Absätze liefert auch einen Hinweis darauf, dass es ausreicht, in einem Suchvorgang nur ein Wort als Akrostichon zu erzeugen.

Ein weiterer Ansatz zur Aufteilung ist die Angabe verschiedener Zeilenlängen bei gleichen Zeilenumbruchfenstergrößen. Beispielsweise kann man die Suche für die Zeilenlängen 30 ± 10, 50 ± 10 und 70 ± 10 durchführen und die durchsuchten Bereiche überschneiden sich höchstens im Suchbereich für die erste Zeile. Ab dem ersten Zeilenumbruch werden drei unabhängige Teilräume durchsucht. In unseren Experimenten (Kapitel 6) wird nur ein Bereich durchsucht, nämlich Zeilenlängen von 50 ± 20 Zeichen.

Bei der verteilten Suche ist es sinnvoll, zur Steigerung der Effizienz die Schwierigkeit der Teilprobleme abzuschätzen und die schwierigeren Probleme zuerst anzugehen. Sollte sich eines der Teilprobleme schon als unlösbar erweisen, ist auch das gesamte Problem nicht lösbar und der Ressourcenaufwand für die Lösung der leichteren Teilprobleme wäre verschwendet. Einen Vorschlag

zur Abschätzung der Schwierigkeit machen wir in Abschnitt 5.4.2.

5.3 Stärkere Heuristik

Bei begrenzten Ressourcen ist es sinnvoll, den Suchprozess intelligent zu steuern, um eine Lösung mit möglichst wenig Aufwand zu erreichen. Das Mittel zur Steuerung der Suche ist eine Heuristik. Wir verwenden die in Abschnitt 4.8 beschriebene CML-Heuristik zur Knotenbewertung. Dabei werden die Knoten bevorzugt expandiert, die näher an einer Lösung liegen. Den Abstand zur Lösung haben wir über die noch nicht erzeugten Akrostichonbuchstaben definiert. Ein Nachteil dieser Methode ist, dass alle Knoten, die auf einem Pfad gleicher Länge erzeugt wurden, und die den gleichen „Buchstabenabstand“ zur Lösung haben auch gleich bewertet werden. Unter gleich bewerteten Knoten wird der nächste zu expandierende Knoten zufällig gewählt. Bei unserer konstruktiven Strategie ist immer der nächste Buchstabe, der erzeugt werden muss, bekannt. Diese Information können wir ausnutzen, indem wir Knoten besser bewerten, bei denen sich der nächste Buchstabe bereits in der Nähe des nächsten potenziellen Zeilenumbruchs befindet. Dazu ermitteln wir, ob ein passender Buchstabe im Bereich der nächsten Zeile verfügbar ist. Wenn dies der Fall ist, können wir den Abstand dieses Buchstabens zur idealen Zeilenumbruchposition, die mit der Zeilenlänge vorgegeben ist, berechnen. Je geringer der Abstand zur idealen Zeilenumbruchposition ist, desto besser wird der Knoten bewertet. Diese Abstandsberechnung fließt in die Heuristik ein und führt zu einer neuen Heuristik, die wir als CML-LD („Constructive Missing Letters - Letter Distance“) bezeichnen.

Ein weiterer Nachteil unserer Methode ist, dass alle Knoten generiert werden müssen und Speicher belegen. Hier bietet es sich an, Operatoren statt Knoten zu bewerten und nur die Operatoren anzuwenden, die mit hoher Wahrscheinlichkeit einen Fortschritt im Suchprozess bringen. Da wir wissen, welcher Buchstabe als nächstes erzeugt werden muss, sollten wir Operatoren bevorzugen, die den Buchstaben mit hoher Wahrscheinlichkeit erzeugen können. Dies sind zunächst die Operatoren, die Zeilentrennungen einfügen, denn nur diese können einen Buchstaben für das Akrostichon direkt erzeugen. Auch bei den Trennoperatoren vermuten wir Unterschiede in der Wahrscheinlichkeit, mit der bestimmte Buchstaben erzeugt werden können. Im folgenden Abschnitt untersuchen wir unter anderem, wie sich diese Wahrscheinlichkeiten verteilen.

5.4 Analyse zur Schwierigkeit der Akrostichonerzeugung

In diesem Abschnitt diskutieren wir, wie die Schwierigkeit der Erzeugung eines Akrostichons zu ermitteln ist. Wie in den letzten Abschnitten angemerkt, ist diese Information auch nützlich, um die Schwierigkeit eines Teilproblems abzuschätzen. Die hierbei ermittelten Daten liefern Ansatzpunkte zur Verbesserung der Suchmethode.

Wir betrachten zwei Faktoren: Die Länge des Akrostichons und die Buchstaben, aus denen es sich zusammensetzt. Wir nehmen an, dass ein Akrostichon schwerer zu erzeugen ist, je länger es ist. Für die Buchstaben nehmen wir an, dass ein Akrostichon, das viele häufige Buchstaben enthält, einfacher zu konstruieren ist, als ein Akrostichon, das eher seltene Buchstaben enthält. Häufige Buchstaben kommen in dem zu verändernden Text häufiger vor und können deshalb mit höherer Wahrscheinlichkeit zur Konstruktion des Akrostichons beitragen als seltene Buchstaben. Eine Analyse der Buchstabenhäufigkeit liefert uns also eine Bewertungsgrundlage für die Schwierigkeit.

Wir wollen auch die Operatoren betrachten, die direkten Einfluss auf die Akrostichonkonstruktion haben - das heißt, die Trennoperatoren. Wir verwenden Zeilentrennoperatoren, die nur zwischen Wörtern trennen, Silbentrennoperatoren und Falschtrennoperatoren, die Wörter an bestimmten Stellen trennen und Absatzoperatoren, die nur zwischen Sätzen trennen dürfen. Wir vermuten, dass sich beispielsweise die Verteilung der Buchstaben am Wortanfang von der Verteilung der Buchstaben am Silbenanfang unterscheidet und dementsprechend ein Zeilenumbruchoperator einen bestimmten Buchstaben mit einer anderen Wahrscheinlichkeit erzeugen kann als der Silbentrennperator. Die folgende Analyse der Buchstabenverteilungen bestätigt diese Vermutung.

5.4.1 Analyse der Buchstabenverteilung

Als Datenbasis für die Analyse der Buchstabenverteilung verwenden wir den *Google Web 1T 5-gram*-Korpus [BF06]. Dieser Korpus enthält n-Gramme, das heißt kurze Phrasen von n Wörtern Länge. Im genannten Korpus sind es 1-Gramme bis 5-Gramme. Sie wurden aus allen von Google (im Jahr 2006) indizierten englischsprachigen Webseiten extrahiert. Dabei wurden Markupinformationen wie HTML-Tags entfernt. Zu jedem n-Gramm ist die Häufigkeit des Auftretens, gezählt über alle indizierten Webseiten, angegeben. Außerdem wurden in den Ausgangsdaten Satzgrenzen identifiziert und Markierungen für Satzbeginn und -ende eingefügt. Der Korpus enthält viele Einträge mit Zahlen, Satz- oder Sonderzeichen. Da wir nur an Buchstaben interessiert sind, haben wir die Daten gesäubert. Wir haben alle Einträge in Kleinbuchstaben umge-

wandelt. Wenn dabei identische Einträge entstanden, wurden die Häufigkeiten addiert. Dann wurden alle Einträge, die nicht ausschließlich aus Kleinbuchstaben und dem Leerzeichen bestanden, entfernt. Eine Ausnahme haben wir bei den 2-Grammen für Satzbeginnmarkierungen $\langle \mathsf{S} \rangle$ gemacht, denn diese geben uns die Möglichkeit, die Verteilung von Buchstaben am Satzbeginn zu untersuchen.

Zur Analyse der Buchstabenverteilungen verwenden wir größtenteils die 1-Gramme, das heißt die Liste der einzelnen Wörter, die das Vokabular des n-Gramm Korpus bilden. Nur für die Satzanfangsbuchstabenverteilung betrachten wir die 2-Gramme. Wir berechnen zunächst die Buchstabenhäufigkeit $freq$ (frequency) und im Anschluss eine Wahrscheinlichkeitsverteilung P (probability distribution). Wir bezeichnen einen Buchstaben mit c (character). Wir wollen die Verteilung der Buchstaben an verschiedenen Positionen untersuchen und verwenden dafür die folgenden Bezeichnungen:

- P_{ap} (a priori) für die positionsunabhängige Wahrscheinlichkeit. Diese Verteilung sollte der allgemeinen Buchstabenwahrscheinlichkeit der englischen Sprache entsprechen.
- P_{wb} (word begin) für die Verteilung der Buchstaben am Wortanfang.
- P_{hy} (hyphenation) für die Verteilung der ersten Buchstaben einer Silbe bei korrekter Silbentrennung (jedoch nicht der Anfangsbuchstaben eines Wortes).
- P_{he} (hyphenation error) für die Verteilung der ersten Buchstaben einer Silbe bei fehlerhafter Silbentrennung, wie in Abschnitt 3.4.5 beschrieben (jedoch nicht der Anfangsbuchstaben eines Wortes).
- P_{sb} für die Verteilung der Buchstaben am Anfang eines Satzes.

Die Berechnung der Häufigkeiten von Buchstaben $freq(c)$ auf Basis der Häufigkeit von Wörtern $freq(w)$ aus dem 1-Gramm-Korpus wird in den folgenden Absätzen erläutert.

Buchstabenverteilung Die Buchstabenhäufigkeit $f_{ap}(c)$ wurde berechnet, indem für jedes Wort w, das den Buchstaben c enthält die Worthäufigkeit $freq(w)$ mit der Anzahl der Vorkommen $count(c, w)$ multipliziert und diese Werte aufsummiert wurden: $f_{ap}(c) = \sum_w freq(w) * count(c, w)$.

Wortanfangsbuchstaben Die Wortanfangsbuchstabenhäufigkeit $f_{wb}(c)$ wurde berechnet, indem die Häufigkeit aller Wörter aufsummiert wurde, die mit c beginnen: $f_{wb}(c) = \sum_w freq(w)\ \forall w : w$ beginnt mit c.

Silbentrennung Alle Wörter wurden in Silben sil aufgeteilt. Dazu wurde der gleiche Algorithmus verwendet, auf dem die Silbentrennoperatoren basieren: $w \rightarrow [sil_0, \ldots, sil_n]$. Die jeweils erste Silbe wurde verworfen, da deren Anfangsbuchstabe auch der Anfangsbuchstabe des Wortes ist und hier nur die Buchstaben nach Trennstellen gezählt werden sollen. Im Anschluss wurde jeder der übrigen Silben die Häufigkeit des Ursprungswortes zugewiesen: $freq(sil_1) = \ldots = freq(sil_n) = freq(w)$. Die Buchstabenhäufigkeit $f_{hy}(c)$ wurde berechnet, indem die Häufigkeit aller Silben aufsummiert wurde, die mit c beginnen: $f_{hy}(c) = \sum_{sil} freq(sil)\ \forall sil : sil$ beginnt mit c.

Falschtrennung Alle Wörter wurden mit dem Falschtrennungsalgorithmus in „falsche Silben“ $fsil$ aufgeteilt. Dazu wurde die gleiche Methode wie bei der Berechnung der Falschtrennoperatoren verwendet (dabei werden nur Trennstellen an den Positionen im Wort erzeugt, wo die korrekte Silbentrennung nicht trennen würde). Das weitere Vorgehen entspricht der Berechnung bei der Silbentrennung: Die jeweils erste $fsil$ jedes Wortes wird verworfen, jeder weiteren wird die Häufigkeit des Wortes zugewiesen. Die Buchstabenhäufigkeit berechnet sich durch $f_{he}(c) = \sum_{fsil} freq(fsil)\ \forall fsil : fsil$ beginnt mit c.

Satzanfangsbuchstaben Zur Berechnung der Häufigkeit von Buchstaben am Satzanfang $f_{sb}(c)$ wurden zunächst alle 2-Gramme ermittelt, die aus dem Satzbeginnsymbol $\langle \mathtt{S} \rangle$ und einem darauf folgenden Wort w_{sb} zusammengesetzt sind. Dem Wort w_{sb} wurde die Häufigkeit des 2-Gramms zugewiesen. Für einen Buchstaben c wurden dann die Häufigkeiten aller w_{sb} aufsummiert die mit c beginnen: $f_{sb}(c) = \sum_{w_{sb}} freq(w_{sb})\ \forall w_{sb} : w_{sb}$ beginnt mit c.

Basierend auf den so ermittelten Häufigkeiten berechnen wir die Wahrscheinlichkeit p (probability) eines Buchstabens, indem die Häufigkeit des Buchstabens durch die Summe aller Buchstabenhäufigkeiten dividiert wird:

$$p(c) = \frac{freq(c)}{\sum_c freq(c)}.$$

Die Wahrscheinlichkeitsverteilung P für alle Buchstaben an den genannten Positionen sind in Tabelle 5.1 dargestellt. Jede Spalte steht für eine der genannten Verteilungen und ist nach absteigender Wahrscheinlichkeit geordnet. Eine grafische Darstellung der Wahrscheinlichkeitsverteilungen, geordnet nach den Buchstaben, zeigt Abbildung 5.1. Die erste Spalte P_{ap} der Tabelle zeigt, wie erwartet, die typische Buchstabenwahrscheinlichkeitsverteilung für englische Texte. Die zweite Spalte P_{wb}, die Verteilung am Wortanfang, zeigt deutliche Unterschiede bei bestimmten Buchstaben. Das „e“, der häufigste Buchstabe der englischen Sprache, kommt seltener am Anfang eines Wortes vor, ebenso

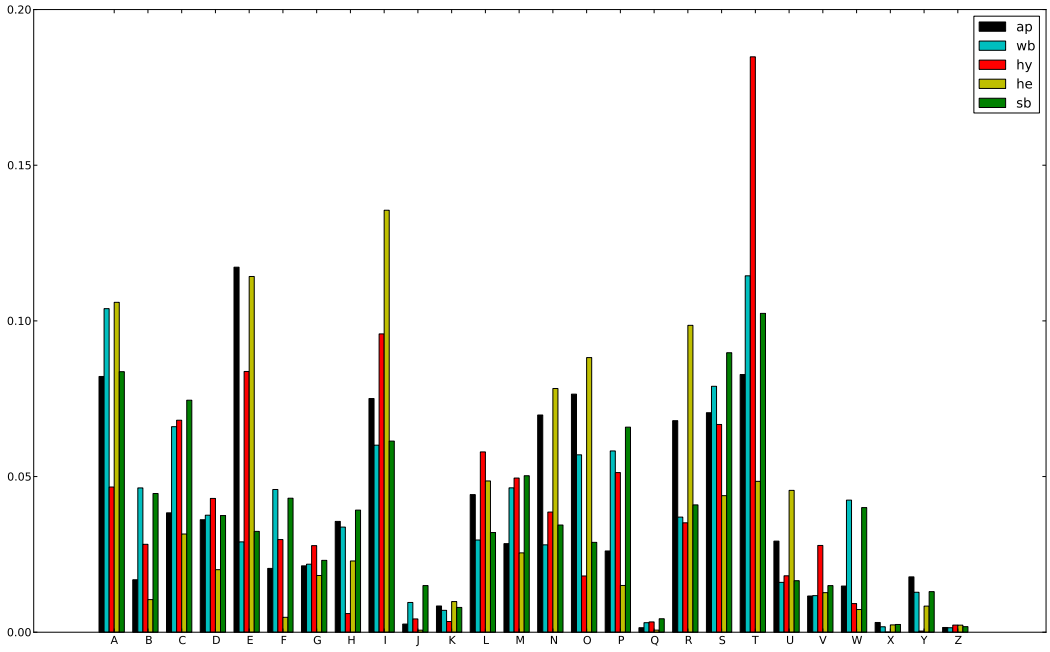


Abbildung 5.1: Wahrscheinlichkeitsverteilung für Buchstaben allgemein (ap), am Wortanfang (wb), nach korrekter (hy) und falscher (he) Silbentrennung und am Satzbeginn (sb).

das „n“, während etwa „c“, „p“ oder „b“ mit höherer Wahrscheinlichkeit am Wortanfang stehen, als sie allgemein vorkommen. Dementsprechend haben wir mit dem Zeilenumbruchoperator bessere Chancen, ein „c“ für das Akrostichon zu erzeugen als ein „e“.

Bei der Verteilung der Silbentrennung oder der Falschtrennung scheinen die Unterschiede zur allgemeinen Buchstabenverteilung weniger stark. Für die Falschtrennung trifft das durchaus zu – die Verteilung bei der korrekten Silbentrennung weicht aber stark ab. Dies äußert sich weniger in der Rangfolge der Buchstaben als in den einzelnen Werten. Besonders die hohe Wahrscheinlichkeit für ein „t“ als ersten Buchstaben nach einer Silbentrennung sorgt für eine starke Abweichung. Der Unterschied zwischen zwei Wahrscheinlichkeitsverteilungen lässt sich auch messen, beispielsweise mit der Kullback-Leibler-Divergenz (KL-Divergenz) [MS99]. Die KL-Divergenz für eine Wahrscheinlichkeitsverteilung P gegenüber einer Wahrscheinlichkeitsverteilung Q ist definiert

als:

$$D_{KL}(P||Q) = \sum_i P(i) \ln \frac{P(i)}{Q(i)}.$$

Nehmen wir als Vergleichsbasis die allgemeine Verteilung der Buchstaben und berechnen die Abweichung der anderen Verteilungen:

$$D_{KL}(P_{wb}||P_{ap}) = 0.1724 \qquad D_{KL}(P_{hy}||P_{ap}) = 0.2008$$
$$D_{KL}(P_{he}||P_{ap}) = 0.0765 \qquad D_{KL}(P_{sb}||P_{ap}) = 0.1891$$

Die Abweichungen für Wortanfänge, Satzanfänge und Silbentrennung sind gegenüber der allgemeinen Buchstabenverteilung recht hoch, während die Falschtrennung eine sehr ähnliche Verteilung hat. Der Falschtrennoperator erzeugt also am wahrscheinlichsten Buchstaben, die der allgemeinen Verteilung entsprechen. Betrachten wir noch den Unterschied zwischen Wort- und Satzanfangsverteilung:

$$D_{KL}(P_{sb}||P_{wb}) = 0.0171$$

Die Verteilung der Buchstaben am Satzanfang entspricht etwa der Verteilung am Wortanfang, was nicht überrascht, denn in beiden Fällen werden Wortanfänge gezählt. Dies weist darauf hin, dass es für die Wortanfangsbuchstabenverteilung unerheblich ist, an welcher Position im Satz sich die Wörter befinden. Eine interessante Abweichung gibt es allerdings beim „o“, einem allgemein häufig auftretenden Buchstaben, der am Wortanfang etwas seltener, am Satzbeginn aber deutlich seltener zu finden ist.

5.4.2 Abschätzung der Konstruktionsschwierigkeit

Basierend auf den im vorherigen Abschnitt vorgestellten Buchstabenverteilungen entwickeln wir nun eine Methode zur Abschätzung der Konstruktionsschwierigkeit eines Akrostichons *ACE* (acrostic construction effort). Wir nehmen an, dass ein Akrostichon schwieriger zu konstruieren ist, je unwahrscheinlicher es „natürlich“ auftritt. Um diese Wahrscheinlichkeit abzuschätzen, nehmen wir an, dass die Buchstaben am Zeilenanfang stochastisch unabhängig voneinander auftreten. Die nur sehr geringe Abweichung der Satzanfangsbuchstabenverteilung P_{sb} von der Wortanfangsbuchstabenverteilung P_{wb} werten wir als Hinweis, dass diese Annahme (zumindest für einen hohen Abstand der Wörter voneinander) begründet ist. Wir betrachten ein Akrostichon A als Menge von Buchstaben c und berechnen die Wahrscheinlichkeit des Akrostichons als Produkt der Buchstabenwahrscheinlichkeiten:

$$p(A) = \prod_{c \in A} p(c).$$

Da wir die „Unwahrscheinlichkeit“ des natürlichen Auftretens eines Akrostichons als Maß für die Konstruktionsschwierigkeit heranziehen wollen, bietet sich die Komplementärwahrscheinlichkeit $1 - p(A)$ an. Die Wahrscheinlichkeiten, die auf diese Weise für Wörter berechnet werden, sind jedoch sehr gering und die Komplementärwahrscheinlichkeiten haben dementsprechend Werte nahe 1. Zum praktischen Umgang mit solchen Werten eignet sich der Logarithmus. Deshalb wollen wir die Konstruktionsschwierigkeit ACE mit dem negativen natürlichen Logarithmus der Wahrscheinlichkeit berechnen:

$$ACE(A) = -\ln p(A).$$

Betrachten wir einige Beispiele: Der Text des Schwarzenegger-Briefs (Abbildung 1.1 auf Seite 5) enthält das Akrostichon „fuckyou“. Auf Basis der allgemeinen Buchstabenverteilung der englischen Sprache berechnen wir die Wahrscheinlichkeit $p_{ap}(\text{fuckyou}) = 7{,}70 * 10^{-12}$, dass dieses Akrostichon natürlich vorkommt, und eine Konstruktionsschwierigkeit von $ACE_{ap}(\text{fuckyou}) = 25{,}59$. Da zur Konstruktion des Akrostichons keine Silbentrennung oder Falschtrennung zum Einsatz kommt, bietet die Verteilung der Buchstaben am Wortanfang möglicherweise eine bessere Abschätzung. In diesem Fall ist die Wahrscheinlichkeit noch etwas geringer ($p_{wb}(\text{fuckyou}) = 3{,}99 * 10^{-12}$) und die entsprechende Konstruktionsschwierigkeit höher: $ACE_{wb}(\text{fuckyou}) = 26{,}25$.

Wenn wir diese Methode verwenden, um die Schwierigkeit für die beiden Wörter einzeln zu bestimmen, ergibt sich erwartungsgemäß ein geringerer Wert für das kürzere Wort: $ACE_{ap}(\text{you}) = 10{,}13$ gegenüber $ACE_{ap}(\text{fuck}) = 15{,}46$. Interessanter ist die Betrachtung gleich langer Wörter, beispielsweise „hello“ und „world“. Beide sind fünf Buchstaben lang, die Berechnung ergibt $ACE_{ap}(\text{hello}) = 14{,}29$ und $ACE_{ap}(\text{world}) = 15{,}91$. Bei einem Problemreduktionsansatz, wie in Abschnitt 5.2 beschrieben, wäre es also effizient, zunächst „world“ zu konstruieren, da „hello“ einfacher ist. Nicht in allen Fällen ergibt sich für längere Wörter auch eine höhere Schwierigkeit. Der acht Buchstaben lange Vorname des Autors „christof“ beispielsweise wird als einfacher zu konstruieren eingeschätzt als das sieben Buchstaben lange Akrostichon „fuckyou“ im Schwarzenegger-Brief: $ACE_{ap}(\text{christof}) = 23{,}48$, da der Name aus häufigeren Buchstaben besteht.

Mit dem ACE haben wir eine einfache Möglichkeit, die Konstruktionsschwierigkeiten verschiedener Akrosticha zu bewerten. Ob dieses Maß wirklich eine gute Abschätzung liefert, kann aber erst eine ausführliche Evaluierung zeigen. Wir sehen auch einige Aspekte, die zur Entwicklung einer besseren Abschätzung nötig wären. Zum einen vermuten wir, dass die Schwierigkeit sehr stark vom ersten Buchstaben des Akrostichons abhängt. Wir haben Experimente durchgeführt, die diese Annahme bestätigen (siehe Abschnitt 6.4).

Eine Möglichkeit, die Schätzung zu verbessern wäre dann, dem ersten Buchstaben ein höheres Gewicht zu geben. Zum anderen ist der Wertebereich des *ACE* nach oben hin unbegrenzt und das Maß an sich nicht normalisiert. Außerdem betrachten wir nur das Akrostichon und inorieren den Text, in dem es konstruiert werden soll. Damit haben wir zwar eine generelle Aussage für die Schwierigkeit, ein bestimmtes Akrostichon in einem beliebigen englischen Text zu konstruieren – interessant ist aber auch die Einschätzung der Schwierigkeit, ein bestimmtes Akrostichon in einem bestimmten Text zu konstruieren. Daraus ergeben sich weitere Fragen, beispielsweise ob bestimmte Eigenschaften von Texten definieren sind, die die Akrostichonkonstruktion ganz allgemein erleichtern oder erschweren. Wir betrachten all diese Aspekte als interessante Punkte für zukünftige Arbeiten.

	P_{ap}		P_{wb}		P_{hy}		P_{he}		P_{sb}
e	0,1173	t	0,1145	t	0,1848	i	0,1355	t	0,1024
t	0,0828	a	0,1039	i	0,0958	e	0,1143	s	0,0898
a	0,0821	s	0,0790	e	0,0838	a	0,1060	a	0,0837
o	0,0765	c	0,0660	c	0,0681	r	0,0986	c	0,0745
i	0,0751	i	0,0601	s	0,0667	o	0,0882	p	0,0659
s	0,0705	p	0,0582	l	0,0579	n	0,0783	i	0,0614
n	0,0698	o	0,0570	p	0,0513	l	0,0486	m	0,0503
r	0,0679	m	0,0464	m	0,0495	t	0,0485	b	0,0446
l	0,0442	b	0,0463	a	0,0466	u	0,0456	f	0,0430
c	0,0383	f	0,0458	d	0,0430	s	0,0439	r	0,0409
d	0,0361	w	0,0424	n	0,0386	c	0,0315	w	0,0400
h	0,0356	d	0,0376	r	0,0351	m	0,0255	h	0,0392
u	0,0293	r	0,0370	f	0,0298	h	0,0229	d	0,0375
m	0,0284	h	0,0338	b	0,0282	d	0,0201	n	0,0344
p	0,0261	l	0,0296	v	0,0279	g	0,0182	e	0,0324
g	0,0213	e	0,0290	g	0,0278	p	0,0150	l	0,0320
f	0,0205	n	0,0281	u	0,0181	v	0,0127	o	0,0289
y	0,0178	g	0,0218	o	0,0181	b	0,0105	g	0,0231
b	0,0168	u	0,0160	w	0,0092	k	0,0098	u	0,0165
w	0,0148	y	0,0128	h	0,0060	y	0,0084	j	0,0150
v	0,0116	v	0,0117	j	0,0043	w	0,0073	v	0,0150
k	0,0084	j	0,0096	k	0,0034	f	0,0048	y	0,0130
x	0,0031	k	0,0070	q	0,0033	x	0,0024	k	0,0080
j	0,0026	q	0,0031	z	0,0023	z	0,0023	q	0,0043
z	0,0015	x	0,0018	y	0,0004	q	0,0007	x	0,0025
q	0,0014	z	0,0014	x	0,0000	j	0,0007	z	0,0018

Tabelle 5.1: Wahrscheinlichkeitsverteilung für Buchstaben allgemein (1. Spalte), am Wortanfang (2. Spalte), nach korrekter (3. Spalte) und falscher (4. Spalte) Silbentrennung und am Satzbeginn (5. Spalte). Die Verteilungen sind jeweils nach absteigender Wahrscheinlichkeit geordnet.

Kapitel 6

Experimente und Evaluation

Dieses Kapitel beschreibt unsere Experimente und deren Ergebnisse. Das wichtigste Ziel der Experimente ist, einen Machbarkeitsnachweis für unsere Methode zur Erzeugung eines Akrostichons zu liefern. Wir gehen davon aus, dass wir nicht jedes beliebige Akrostichon in jedem beliebigen Text erzeugen können und wollen ermitteln, wie erfolgreich unser Verfahren im Allgemeinen ist. Außerdem wollen wir unsere Vermutung überprüfen, dass der erste Buchstabe kritisch für den Erfolg der gesamten Akrostichonerzeugung ist (vgl. Abschnitt 3.12).

Da wir das erste Verfahren dieser Art entwickeln und somit keine Vergleichssysteme existieren, sollen die Ergebnisse als Basiswerte für zukünftige Verbesserungen des Suchverfahrens und als technischer Benchmark dienen.

6.1 Testkorpus

Unser Testkorpus besteht aus zwei Teilen: Texte und Zielakrosticha, die in den Texten erzeugt werden sollen. Die Texte sollten möglichst alltäglicher Art sein, etwa Nachrichtenartikel, Emails oder Blogeinträge. Wir haben eine Auswahl aus zwei bekannten Korpora erstellt. Der Auswahlprozess wird in Abschnitt 6.1.1 beschrieben. Wir haben die Auswahl auf die geringe Anzahl von 50 Texten beschränkt, da wir es für wichtiger hielten, eine größere Anzahl Akrosticha zu testen. In den Experimenten wird jedes Zielakrostichon mit jedem Text kombiniert, und uns steht nur begrenzte Zeit für die Experimente zur Verfügung. Bei den Akrosticha unterscheiden wir „natürliche“ Zielakrosticha, wie etwa Namen und „künstliche“ Zielakrosticha, wie etwa Einzelbuchstabensequenzen. Die Auswahl der Zielakrosticha wird in Abschnitt 6.1.2 beschrieben.

6.1.1 Texte

Wir haben einen Korpus von 50 Texten zu gleichen Teilen aus zwei Arten alltäglicher Texte konstruiert: Nachrichtenartikel und Emails.

Die Artikel wurden aus dem Reuters Corpus Volume 1 (RCV1) [LYR+04] gezogen. Der RCV1 ist eine Standard-Testkollektion für Textklassifikation und enthält etwa 810.000 Artikel der Nachrichtenagentur Reuters aus dem Zeitraum August 1996 bis August 1997. Die Artikel sind annotiert und liegen in einem XML-Format vor. Wir sind nur am Text der Artikel und nicht an den Metainformationen interessiert. Aus dem gesamten Korpus wurden 500 Artikel zufällig gezogen sowie der eigentliche Nachrichtentext extrahiert. Der Text enthielt noch HTML-Entities (beispielsweise `&quot;` oder `&amp;`), die aufgelöst, und Absatzmarkierungen, die entfernt wurden. Dann wurden alle Artikel mit weniger als 150 und mehr als 5000 Zeichen (inklusive Leerzeichen) verworfen. Aus den übrigen wurden 25 Artikel manuell gewählt. Dabei wurden Artikel verworfen, die nur Listen oder Tabellen enthielten (beispielsweise Sportergebnisse oder Börsenkurse).

Emails wurden aus dem Enron-Email-Korpus [KY04] gezogen. Der Korpus enthält etwa 500.000 Emails von 153 Mitarbeitern der Firma Enron. Die Emails wurden im Laufe der Untersuchungen eines Bilanzfälschungsskandals veröffentlicht und sind der umfangreichste frei verfügbare Korpus „echter“ Emails, was ihn zu einem wertvollen Studienobjekt etwa zur Spam-Klassifikation oder Netzwerkanalyse macht. Der Korpus ist in Ordnern organisiert, jeweils ein Ordner pro Person. Jeder Ordner enthält typische Unterordner, die von Mailclients zur Verwaltung der Emails angelegt werden, wie Posteingang, Postausgang, Kontakte oder Kalender. Einige der Ordner enthalten Duplikate. Für einen gesäuberten Korpus wird eine Anzahl von etwa 200.000 verschiedenen Emails berichtet [KY04]. Wir haben den Korpus ungesäubert in der Version vom 21.08.2009 vorliegen. Um Duplikate zu vermeiden, haben wir uns auf die Postausgangsordner beschränkt (alle Ordner mit „sent“ im Namen). Die Anzahl der Mails in den Postausgangsordnern beträgt 126.075. Aus diesen wurden 5000 Mails zufällig gezogen und die Mailheader entfernt. Dann wurden alle Mails mit weniger als 150 und mehr als 5000 Zeichen (inklusive Leerzeichen) verworfen und aus den übrigen 25 manuell ausgewählt. Dabei wurde, wie bei den Artikeln, darauf geachtet, keine Texte zu wählen, die nur Listen oder Tabellen enthielten (beispielsweise Preislisten), sowie Antwortmails oder weitergeleitete Mails verworfen. Außerdem wurden, wenn nötig, Signaturen entfernt. Tabelle 6.1 zeigt einige statistische Werte des Textkorpus. Die minimale Textlänge von 150 Zeichen ergibt sich aus fünf Zeilen bei einer Zeilenlänge von 30 Zeichen und bietet damit genügend Platz für ein durchschnittliches englisches Wort als Akrostichon. Der Maximalwert von 5000 Zeichen ergibt sich aus 50

Länge (Zeichen)	Anzahl
<1.000	26
1.000-2.000	15
2.000-3.000	4
3.000-4.000	4
>4.000	1
Gesamt	50

Tabelle 6.1: Verteilung der Dokumentlängen der Experimentkorpustexte. Die minimale Länge beträgt 197, die maximale Länge 4.223 und die durchschnittliche Länge 1.265,8 Zeichen.

Zeilen bei einer durschnittlichen Zeilenlänge von 100 Zeichen und füllt etwa eine Bildschirmseite.

6.1.2 Zielakrosticha

Um verschiedene Aspekte der Akrostichonerzeugung zu untersuchen haben wir Sammlungen natürlicher und künstlicher Zielakrosticha zusammengestellt. Als natürliche Akrosticha betrachten wir „normale“ Wörter, Phrasen und Sätze wie sie im üblichen Sprachgebrauch verwendet werden. Da wir nicht die Ressourcen haben, etwa ein ganzes Wörterbuch zu überprüfen, haben wir uns entschlossen, häufige Wörter und Vornamen als Vertreter der natürlichen Akrosticha zu wählen. Als künstliche Akrosticha betrachten wir beispielsweise Buchstabensequenzen. Die Auswahl wird im folgenden beschrieben.

Häufige Wörter Wir verwenden Listen[1] der häufigsten englischen Wörter verschiedener Wortarten (Substantive, Verben, Adjektive, Präpositionen). Außerdem ergänzen wir diejenigen der 100 häufigsten englischen Wörter, die nicht bereits in einer der genannten Listen vorkommen. Die Wörter sind in den Tabellen A.2 und A.3 dargestellt. Es sind je 25 Substantive, Verben und Adjektive und 17 Präpositionen. Unter den 100 allgemein häufigsten englischen Wörtern kommen bereits 39 in diesen vier Wortarten vor. Die restlichen 61 haben wir als „Sonstige“ mit in den Korpus aufgenommen. Damit umfasst dieser Teil des Korpus 153 Zielakrosticha.

[1] Quelle: `http://en.wikipedia.org/wiki/Common_english_words`, letzter Abruf 22.8.2012

Vornamen Namen sind klassische Anwendungsfälle für Akrosticha (zumindest in der Poesie). Wir verwenden die jeweils 50 häufigsten amerikanischen Frauen- und Männernamen[2] des 20. Jahrhunderts sowie die jeweils 50 häufigsten deutschen Frauen- und Männernamen[3] des Jahres 2010. Die deutschen Vornamenslisten enthalten auf Grund eines Fehlers in der Vorverarbeitung teilweise verschiedene Schreibweisen eines Namens, was allerdings nicht sehr problematisch ist: Zum einen sind es verschiedene Buchstaben und zum anderen können wir bei dieser Gelegenheit Spezialfälle betrachten, etwa wenn sich „Lucas“ erzeugen lässt, nicht aber „Lukas“. Die Namen sind in Tabelle A.1 dargestellt. Dieser Teil des Korpus umfasst 200 Zielakrosticha.

Damit umfasst der Anteil natürlicher Zielakrosticha 353 Wörter. Weitere geeignete Akrosticha für diese Kategorie wären kurze Phrasen wie Redewendungen oder Zitate, oder ganze Sätze. Besondere Sätze, die wir in Betracht gezogen haben sind Pangramme, das heißt Sätze, die jeden Buchstaben mindestens einmal enthalten (ein bekanntes Beispiel ist „The quick brown fox jumps over the lazy dog“). Wir haben auch hier Listen zusammengestellt. Sie kommen in den Experimenten jedoch noch nicht zum Einsatz, da wir sie als zu lang einschätzen, um sie mit dem Basisverfahren erfolgreich zu erzeugen. Sie sind geeignet, in Zukunft ein verbessertes Verfahren wie etwa das in Abschnitt 5.2 beschriebene Problemreduktionsverfahren zu evaluieren.

Künstliche Akrosticha Wir untersuchen, wie gut wir Sequenzen identischer Buchstaben („aaaa“, „bbbb“, usw.) (*SIB*) sowie das Alphabet von „a“ bis „z“ (*Abc*) und umgekehrt (*Zyx*) als Akrostichon erzeugen können. Wir betrachten hierbei nur die 26 Kleinbuchstaben. Für diese Experimente verwenden wir keine Listen, sondern einen speziellen Experimentaufbau, der in Abschnitt 6.2 beschrieben wird.

Reflektives Akrostichon Als „reflektives Akrostichon“ bezeichnen wir das Akrostichon, das den Text, in dem es erzeugt wird, widergibt. Ein Beispiel dafür ist der kurze Demonstrationstext aus Kapitel 3:

```
Hello world is an easy
example program to
learn a new programming
language before writing
other, more complex programs.
```

[2] Quelle: `http://www.ssa.gov/OACT/babynames/decades/century.html`, letzter Abruf 22.8.2012

[3] Quelle: `http://www.beliebte-vornamen.de/jahrgang/j2010/top500-2010`, letzter Abruf 22.8.2012

Der Textanfang „Hello“ ist gleichzeitig das Akrostichon. Bei dieser Variante besteht die Gewissheit, dass der erste Buchstabe immer passt. Wir können also überprüfen, ob der erste Buchstabe besonders wichtig für den Erfolg bei der Erzeugung des gesamten Akrostichons ist, wie wir in Abschnitt 3.12 vermutet haben. Die Erfolgsquote bei der Konstruktion des reflektiven Akrostichons sollte deutlich höher liegen als bei den anderen Akrosticha. Auch hierfür verwenden wir einen speziellen Experimentaufbau.

6.2 Aufbau der Experimente

In jedem der 50 Texte wird versucht, jedes der 353 natürlichen Akrosticha zu erzeugen. Damit ergibt sich die Anzahl von 17.650 Experimenten. Wir erfassen für jedes Experiment die folgenden Kennwerte:

- Anzahl der generierten Knoten (*nodes*)
- Generierte Knoten pro Sekunde (generated nodes per second, *gnps*)
- Anzahl der Zieltests (goal checks, *gc*)
- Zieltests pro Sekunde (goal checks per second, *gcps*)

Für erfolgreiche Suchvorgänge wird zusätzlich die Sequenz aller angewendeten Operatoren, beziehungsweise deren Anzahl (*pathlen*) erfasst.

Für die künstlichen Akrosticha wurden spezielle Experimente durchgeführt. Beim Alphabet-Experiment wird zunächst im ersten Text versucht, „a“ als Akrostichon zu erzeugen. Nur im Erfolgsfall wird daraufhin „ab“ versucht, dann „abc“ usw. Schlägt ein Versuch fehl, wird mit dem nächsten Text fortgefahren. Dies wird wiederholt, bis der Versuch für alle 50 Texte einmal fehlgeschlagen ist. Das gleiche Experiment wird mit dem umgekehrten Alphabet durchgeführt.

Das Buchstabensequenz-Experiment ist ähnlich aufgebaut. Im ersten Text wird versucht, den Buchstaben „a“ als Akrostichon zu erzeugen. Bei Erfolg wird versucht, „aa“ zu erzeugen usw., bis ein Versuch fehlschlägt. Dann wird mit dem Buchstaben „b“ ebenso verfahren. Sind alle Buchstaben bis „z“ einmal durchlaufen, wird der Prozess für alle weiteren Texte wiederholt.

Auch das Reflektives-Akrostichon-Experiment wird für jeden Text einmal durchgeführt. Der Text wird eingelesen und normalisiert, damit er vom Suchsystem als Akrostichon akzeptiert wird. Das heißt, es werden alle Buchstaben zu Kleinbuchstaben konvertiert und daraufhin alle Zeichen, die keine Kleinbuchstaben sind, verworfen. In diesem Schritt werden auch Leerzeichen entfernt,

die wir mit dem aktuellen System noch nicht erzeugen können. Dann wird versucht, im Originaltext die ersten beiden Buchstaben (mit dem ersten Buchstaben allein zu beginnen ist unnötig, er steht ja schon da) des normalisierten Textes als Akrostichon zu erzeugen. Im Erfolgsfall werden die ersten drei Buchstaben des normalisierten Textes probiert usw., bis ein Versuch fehlschlägt.

6.3 Konfiguration des Testsystems

Für die Experimente verwenden wir eine Best-First-Suche mit der in Abschnitt 4.8 beschriebenen CML-Heuristik. Die eingesetzten Operatoren sind: Context-Synonym-First und -Last für Phrasen der Länge 3, 4 und 5, Context-Synonym-Center für Phrasen der Länge 3 und 5, Context-Prepend und Context-Append für Phrasen der Länge 2, 3 und 4, Context-Insert für Phrasen der Länge 2 und 4, Funktionsworte, Rechtschreibfehler, Kontraktion und Expansion sowie Zeilentrennung, Absatztrennung, Silbentrennung und Falschtrennung. Die Zeilenlänge wurde auf 50 Zeichen und das Zeilenumbruchfenster auf 20 Zeichen festgelegt, was Zeilenlängen zwischen 30 und 70 Zeichen erlaubt.

Pilotexperimente ergaben, dass erfolglose Suchvorgänge im Durchschnitt fünf bis zehn Minuten dauern, bis die Suche wegen Speichermangel abgebrochen wird. Da wir mit vielen erfolglosen Suchvorgängen rechnen, wurde die maximale Anzahl expandierter Knoten, das heißt die Anzahl der Zieltests, auf 50.000 begrenzt. Dieser Wert reduziert die maximale Suchdauer im Durchschnitt auf unter eine Minute. Wir vermuten, dass die Masse der erfolgreichen Experimente mit weniger als 50.000 Zieltests auskommt, weil die Experiment-Akrosticha recht kurz sind, so dass diese Einschränkung kein Problem darstellt.

Die Experimente wurden größtenteils auf handelsüblichen Rechnern (Intel Core2 Quad Q9550, 2.83GHz, 16GB RAM) durchgeführt.

6.4 Ergebnisse

Die Ergebnisse der Experimente mit natürlichen Akrosticha sind in zwei Tabellen dargestellt. Tabelle 6.2 zeigt die Auswertung für alle Experimente und gibt Auskunft über die Erfolgsquote, Statistische Werte der Akrosticha und Performanz des Systems. Wir sind in der Lage, in 13,87% der Experimente das Akrostichon zu erzeugen, ein Wert, den wir als sehr gut für das Basissystem betrachten. Das System generiert im Durchschnitt 148.041 Knoten pro Sekunde und ist in der Lage, durchschnittlich 1.369 Zieltestes pro Sekunde durchzuführen. Das Verhältnis von generierten Knoten zu Zieltests beträgt mehr als 108:1. Dieser Wert weist auf viel Optimierungspotenzial hin. Ansätze

	Anzahl						
Typ	Gesamt	Erfolg	Quote	∅*len*	∅*ACE*	∅*gcps*	∅*gnps*
Adjektive	1.250	155	12,40%	4,64	13,84	1.815	173.883
Präpositionen	850	193	22,71%	3,53	10,43	1.262	116.314
Substantive	1.250	164	13,12%	4,76	14,40	1.352	127.989
Verben	1.250	204	16,32%	3,60	10,93	1.294	123.391
Sonstige	3.050	680	22,30%	3,36	9,92	1.531	143.086
Namen (de, m)	2.500	234	9,36%	5,06	15,58	1.369	129.731
Namen (de, w)	2.500	285	11,40%	4,78	13,81	1.649	156.732
Namen (us, m)	2.500	229	9,16%	6,08	18,56	1.072	166.800
Namen (us, w)	2.500	304	12,16%	6,34	18,35	1.049	165.168
Gesamt	**17.650**	**2.448**	**13,87%**	**4,82**	**14,38**	**1.369**	**148.041**

Tabelle 6.2: Auswertung aller Experimente mit natürlichen Akrosticha. Angegeben ist die Anzahl der Experimente und der Anteil der erfolgreichen Experimente sowie Durchschnittswerte für die Länge der Akrosticha (textitlen), die Konstruktionsschwierigkeit (*ACE*), die Zieltestes pro Sekunde (*gcps*) und die generierten Knoten pro Sekunde (*gnps*).

zur Optimierung sind hier eine Operator- statt Knotenauswahl und eine verbesserte Heuristik zu Bewertung der Operatoren, wie in Abschnitt 5.3 vorgestellt.

In Tabelle 6.3 wird der erfolgreiche Anteil der Experimente genauer untersucht. Die durchschnittliche Anzahl von 2.307 Zieltestes für lösbare Akrosticha zeigt, dass die Begrenzung auf maximal 50.000 Zieltests (vgl. Abschnitt 6.3) die Ergebnisse nicht stark beeinflusst. Unter den lösbaren Akrosticha finden sich im Durchschnitt die kürzeren Wörter. Die durchschnittliche Länge beträgt 4,36 Buchstaben gegenüber 4,82 bei allen getesteten Wörtern. Die durchschnittliche Länge eines Lösungspfades beträgt 4,93, das heißt für weniger als 20% aller zu erzeugenden Buchstaben ist mehr als eine Operatoranwendung notwendig, und die meisten werden in einem Schritt erzeugt. Eine Übersicht über die verwendeten Operatoren gibt Tabelle 6.4. Die Rangfolge in der Tabelle kann als Stärke der Operatoren interpretiert werden, das heißt als ihr Potenzial schnell zu einer Lösung zu führen (vgl. Abschnitt 3.2). Der häufigste verwendete Operator ist der Falschtrennoperator, der nach unserer Einschätzung zu den stärksten Operatoren gehörte, was sich hier bestätigt.

Die Ergebnisse der Experimente mit künstlichen Akrosticha sind in Tabelle 6.5 dargestellt. Diese Werte zeigen den Einfluss des ersten Buchstabens auf die Erfolgsquote. Am deutlichsten wird dies beim reflektiven Akrostichon, bei dem der erste Buchstabe immer vorhanden ist. In 48 von 50 Texten, das heißt in 96% aller Fälle, konnte mindestens der zweite Buchstabe erzeugt werden, im

Typ	Anzahl	∅*len*	∅*ACE*	∅*gc*	∅*pathlen*
Adjektive	155	4,36	13,06	1.715	5,05
Präpositionen	193	3,44	9,97	1.498	3,99
Substantive	164	4,47	13,50	1.733	5,02
Verben	204	3,59	10,76	1.593	4,14
Sonstige	680	3,27	9,46	1.437	3,76
Namen (de, m)	234	4,45	13,27	1.617	4,93
Namen (de, w)	285	4,85	13,59	1.734	5,32
Namen (us, m)	229	6,00	17,91	4.991	6,76
Namen (us, w)	304	6,07	17,20	4.905	6,83
Gesamt	**2.448**	**4,36**	**12,70**	**2.307**	**4,93**

Tabelle 6.3: Auswertung aller erfolgreichen Experimente mit natürlichen Akrosticha. Angegeben ist die Anzahl der Experimente sowie Durchschnittswerte für die Länge der Akrosticha (textitlen), die Konstruktionsschwierigkeit (*ACE*), die Anzahl der Zieltestes (*gc*) und die Lösungspfadlänge (*pathlen*).

Durchschnitt konnten über sieben Buchstaben erzeugt werden und das längste auf diese Art erzeugte Akrostichon war 20 Buchstaben lang. Ein Akrostichon von sechs Buchstaben Länge konnte in 42 der 50 Texte erzeugt werden, das entspricht einer Erfolgsquote von 84% für die Erzeugung eines Wortes durchschnittlicher Länge in der englischen Sprache.

Das Alphabet beginnend mit „a“, dem dritthäufigsten Buchstaben in englischen Texten (und ebenso der dritthäufigste Buchstabe am Satzbeginn) konnte in 30% der Experimente als Akrostichon erzeugt werden und dann im Durchschnitt bis über den dritten Buchstaben hinaus. Das umgekehrte Alphabet beginnend mit „z“, dem zweitseltensten Buchstaben in englischen Texten (und seltensten Buchstaben am Satzbeginn) konnte hingegen in keinem der 50 Texte erzeugt werden.

Sequenzen identischer Buchstaben konnten in 14,08% aller Fälle erzeugt werden, was etwa der allgemeinen Erfolgsquote bei den natürlichen Akrosticha entspricht und zu erwarten war, da hier jeder Buchstabe einmal mit jedem Text kombiniert wird. Erfolgreiche Sequenzen werden im Durchschnitt bis über den achten Buchstaben hinaus erzeugt. Die längste Sequenz bestand aus 56 Buchstaben, erwartungsgemäß handelt es sich dabei um den Buchstaben „t“. Dieses Ergebnis zeigt, dass unser System durchaus in der Lage ist, längere Akrosticha zu erzeugen.

Anzahl	Operator
4532	Falschtrennoperator
2499	Zeilenumbruchoperator
1644	Context-Prepend-Operator(3)
1161	Silbentrennoperator
1077	Context-Insert-Operator(2)
324	Funktionswortoperator(1)
207	Rechtschreibfehleroperator
207	Context-Synonym-First-Operator(3)
186	Context-Append-Operator(3)
132	Context-Prepend-Operator(4)
57	Context-Synonym-Last-Operator(3)
27	Absatzoperator
9	Context-Append-Operator(4)
8	Context-Synonym-Center-Operator(3)
1	Context-Synonym-First-Operator(4)

Tabelle 6.4: Auswertung der Operatortypen, die in den erfolgreichen Experimenten mit natürlichen Akrosticha angewendet wurden. Der Wert in Klammern gibt bei parametrisierten Operatortypen den Parameter an.

	Anzahl			Akrostichonlänge		
Experiment	Gesamt	Erfolg	Quote	*min*	∅	*max*
Reflektiv	50	48	96,00%	2	7,25	20
Abc	50	15	30,00%	1	3,47	9
Zyx	50	0	0,0%	-	-	-
SIB	1.300	183	14,08%	1	8,51	56

Tabelle 6.5: Auswertung der Experimente mit künstlichen Akrosticha. Die Angabe für erfolgreiche Experimente und die Erfolgsquote betrachten die Experiemente bei denen ein Akrostichon minimaler Länge erzeugt werden konnte. Die Angaben für die Akrostichonlänge beziehen sich nur auf die erfolgreichen Experimente.

Kapitel 7

Zusammenfassung

Diese Arbeit zeigt, wie heuristische Suche eingesetzt werden kann, um ein Akrostichon in einem Text zu erzeugen. Wir haben eine Reihe von Operatoren zur systematischen Textveränderung entwickelt, die in einem Suchverfahren eingesetzt werden, um eine große Menge von Variationen eines Textes zu erstellen. Der so aufgespannte Suchraum wird mit einer Best-First-Suche durchsucht, um eine Textvariation zu finden, die das Akrostichon enthält. Wir haben ein Demonstrationssystem implementiert und evaluiert. Unsere Ergebnisse zeigen, dass dieses System in über 13% aller Fälle ein beliebiges Wort als Akrostichon in einem beliebigen Text erzeugen kann. Wir können weiterhin zeigen, dass diese Erfolgsquote in einem speziellen Fall, nämlich der Erzeugung eines reflexiven Akrostichons (der Text selbst wird als Akrostichon erzeugt), sogar bei 84% liegt (gemessen an der Erzeugung eines mindestens sechs Buchstaben langen Akrostichons). Der Grund für diesen Unterschied ist die besondere Schwierigkeit, den ersten Buchstaben zu erzeugen.

Das vorgestellte System betrachten wir als Machbarkeitsnachweis und die allgemeine Erfolgsquote als Basiswert für zukünftige Entwicklungen. Zur Steigerung der Performanz schlagen wir weitere Operatoren vor, wie etwa Satzbeginnerweiterung oder Grammatikoperatoren und haben außerdem Verbesserungen des Suchverfahrens, wie verteilte Suche und eine stärkere Heuristik diskutiert.

Für zukünftige Untersuchungen sehen wir einige interessante Fragen: Unser System verwendet für alle Operatoren die gleichen Operatorkosten. Wie müssen die Operatorkosten stattdessen konfiguriert werden, um möglichst fehlerfreie Lösungen zu erhalten, oder um den Suchaufwand zu minimieren? Wie können wir Methoden des maschinellen Lernens einsetzen, um die idealen Operatorkosten für bestimmte Zwecke zu ermitteln?

Unsere einfache Betrachtung der Konstruktionsschwierigkeit für ein Akrostichon konzentriert sich nur auf das Akrostichon und beachtet den den Text

nicht. Wie kann eine Analyse des Textes eingesetzt werden, um die Abschätzung für den Konstruktionsaufwand zu verbessern? Welche Eigenschaften muss ein Text aufweisen, damit ein beliebiges Akrostichon mit möglichst hoher Wahrscheinlichkeit darin erzeugt werden kann? Gibt es diesbezüglich Unterschiede zwischen Texten verschiedener Themengebiete, Genres oder Stile?

Die Erzeugung eines Akrostichons durch heuristische Suche ist nur ein Beispiel für die Kombination von Künstlicher Intelligenz und Computerlinguistik. In der Schnittmenge dieser Forschungsgebiete sehen wir großes Potenzial für zukünftige Forschung und Entwicklung. Anwendungsgebiete finden sich im Paraphrasing, in der Anfrageerweiterung für Information Retrieval-Systeme, der maschinellen Übersetzung oder in automatisierten Dialogsystemen.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[Arc99] Aaron F. Archer. A modern treatment of the 15 puzzle. *American Mathematical Monthly*, 106:793–799, 1999.

[BCB05] Colin J. Bannard und Chris Callison-Burch. Paraphrasing with bilingual parallel corpora. In Kevin Knight, Hwee Tou Ng und Kemal Oflazer (Hrsg.), *ACL*. The Association for Computer Linguistics, 2005.

[BF06] Thorsten Brants und Alex Franz. Web 1T 5-gram Version 1, 2006.

[BG04] Igor A. Bolshakov und Alexander F. Gelbukh. Synonymous paraphrasing using wordnet and internet. In Farid Meziane und Elisabeth Métais (Hrsg.), *NLDB*, Band 3136 aus *Lecture Notes in Computer Science*, Seiten 312–323. Springer, 2004.

[BL03] Regina Barzilay und Lillian Lee. Learning to paraphrase: an unsupervised approach using multiple-sequence alignment. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1*, NAACL '03, Seiten 16–23, Stroudsburg, PA, USA, 2003. Association for Computational Linguistics.

[BM01] Regina Barzilay und Kathleen McKeown. Extracting paraphrases from a parallel corpus. In *ACL*, Seiten 50–57. Morgan Kaufmann Publishers, 2001.

[CB08] Chris Callison-Burch. Syntactic constraints on paraphrases extracted from parallel corpora. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '08, Seiten 196–205, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics.

[DB05] William B. Dolan und Chris Brockett. Automatically constructing a corpus of sentential paraphrases. In *Proceedings of the 3rd Int. Workshop on Paraphrasing*, Seiten 9–16, 2005.

[Fel98] Christiane Fellbaum. *WordNet An Electronic Lexical Database*. MIT Press, 5 1998.

[GFW+01] Robert Gaizauskas, Jonathan Foster, Yorick Wilks, John Arundel, Paul Clough und Scott Piao. The meter corpus: A corpus for analysing journalistic text reuse. In *The Corpus Linguistics 2001 Conference*, Seiten 214–223, 2001.

[HNR68] Peter E. Hart, Nils J. Nilsson und Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[HNR72] Peter E. Hart, Nils J. Nilsson und Bertram Raphael. Correction to „a formal basis for the heuristic determination of minimum cost paths“. *SIGART Bull.*, (37):28–29, Dezember 1972.

[KB06] David Kauchak und Regina Barzilay. Paraphrasing for automatic evaluation. In Robert C. Moore, Jeff A. Bilmes, Jennifer Chu-Carroll und Mark Sanderson (Hrsg.), *HLT-NAACL*. The Association for Computational Linguistics, 2006.

[Knu91] Donald E. Knuth. *The TeXbook*. Addison-Wesley, 20. Auflage, May 1991.

[KY04] Bryan Klimt und Yiming Yang. The enron corpus: A new dataset for email classification research. In Jean-François Boulicaut, Floriana Esposito, Fosca Giannotti und Dino Pedreschi (Hrsg.), *Machine Learning: ECML 2004*, Band 3201 aus *Lecture Notes in Computer Science*, Seiten 217–226. Springer Berlin / Heidelberg, 2004.

[LYR+04] David D. Lewis, Yiming Yang, Tony G. Rose, Fan Li, G. Dietterich und Fan Li. Rcv1: A new benchmark collection for text categorization research. *Journal of Machine Learning Research*, 5:361–397, 2004.

[MH11] Donald Metzler und Eduard Hovy. Mavuno: a scalable and effective hadoop-based paraphrase acquisition system. In *Proceedings of the Third Workshop on Large Scale Data Mining: Theory and Applications*, LDMTA '11, Seiten 3:1–3:8, New York, NY, USA, 2011. ACM.

[MHZ11] Donald Metzler, Eduard H. Hovy und Chunliang Zhang. An empirical evaluation of data-driven paraphrase generation techniques. In *ACL (Short Papers)*, Seiten 546–551. The Association for Computer Linguistics, 2011.

[MS99] Christopher D. Manning und Hinrich Schuetze. *Foundations of Statistical Natural Language Processing*. The MIT Press, 1. Auflage, June 1999.

[Nav09] Roberto Navigli. Word sense disambiguation: A survey. *ACM Comput. Surv.*, 41(2), 2009.

[PD05] Marius Paşca und Péter Dienes. Aligning needles in a haystack: Paraphrase acquisition across the web. In Robert Dale, Kam-Fai Wong, Jian Su und Oi Kwong (Hrsg.), *Natural Language Processing - IJCNLP 2005*, Band 3651 aus *Lecture Notes in Computer Science*, Seiten 119–130. Springer Berlin / Heidelberg, 2005.

[Pea84] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison Wesley, 1984.

[PKM03] Bo Pang, Kevin Knight und Daniel Marcu. Syntax-based alignment of multiple translations: extracting paraphrases and generating new sentences. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1*, NAACL '03, Seiten 102–109, Stroudsburg, PA, USA, 2003. Association for Computational Linguistics.

[PTS10] Martin Potthast, Martin Trenkmann und Benno Stein. Netspeak: Assisting Writers in Choosing Words. In Cathal Gurrin, Yulan He, Gabriella Kazai, Udo Kruschwitz, Suzanne Little, Thomas Roelleke, Stefan M. Rüger und Keith van Rijsbergen (Hrsg.), *Advances in Information Retrieval. 32nd European Conference on Information Retrieval (ECIR 10)*, Band 5993 aus *Lecture Notes in Computer Science*, Seite 672, Berlin Heidelberg New York, 2010. Springer.

[Rei93] Alexander Reinefeld. Complete solution of the eight-puzzle and the benefit of node ordering in ida*. In Ruzena Bajcsy (Hrsg.), *IJCAI*, Seiten 248–253. Morgan Kaufmann, 1993.

[RN10] Stuart Russell und Peter Norvig. *Artificial Intelligence: A Modern Approach, 3rd Edition*. Prentice Hall, 3. Auflage, December 2010.

[Rus92] Stuart Russell. Efficient memory-bounded search methods. In *In ECAI-92*, Seiten 1–5. Wiley, 1992.

[RW86] Daniel Ratner und Manfred K. Warmuth. Finding a shortest solution for the n × n extension of the 15-puzzle is intractable. In *AAAI*, Seiten 168–172, 1986.

[Tre08] Martin Trenkmann. NETSPEAK - Ein Assistent zum Verfassen fremdsprachiger Texte. Bachelorarbeit, Bauhaus-Universität Weimar, Fakultät Medien, Medieninformatik, Juni 2008.

[ZH02] Rong Zhou und Eric A. Hansen. Memory-bounded a* graph search. In Susan M. Haller und Gene Simmons (Hrsg.), *FLAIRS Conference*, Seiten 203–209. AAAI Press, 2002.

Anhang A

Zielakrosticha des Experimentkorpus

amerikanische Vornamen (20. Jh)		deutsche Vornamen (2010)	
weiblich	männlich	weiblich	männlich
Mary	James	Mia	Leon
Patricia	John	Hannah	Lucas
Elizabeth	Robert	Hanna	Lukas
Jennifer	Michael	Lena	Ben
Linda	William	Lea	Finn
Barbara	David	Leah	Fynn
Susan	Richard	Emma	Jonas
Margaret	Joseph	Anna	Paul
Dorothy	Charles	Leonie	Luis
Jessica	Thomas	Leoni	Louis
Sarah	Christopher	Lilli	Maximilian
Betty	Daniel	Lilly	Luca
Nancy	Matthew	Lili	Luka
Karen	Donald	Emilie	Felix
Lisa	Anthony	Emily	Tim
Helen	Paul	Lina	Timm
Sandra	Mark	Laura	Elias
Donna	George	Marie	Max
Ashley	Steven	Sarah	Noah
Kimberly	Kenneth	Sara	Philip
Carol	Andrew	Sophia	Philipp
Michelle	Edward	Sofia	Niclas
Amanda	Brian	Lara	Niklas

amerikanische Vornamen (20. Jh)		deutsche Vornamen (2010)	
weiblich	männlich	weiblich	männlich
Emily	Joshua	Sophie	Julian
Melissa	Kevin	Sofie	Moritz
Laura	Ronald	Maja	Jan
Deborah	Timothy	Maya	David
Stephanie	Jason	Amelie	Fabian
Rebecca	Jeffrey	Luisa	Alexander
Sharon	Gary	Louisa	Simon
Ruth	Ryan	Johanna	Jannik
Cynthia	Eric	Emilia	Yannik
Kathleen	Nicholas	Nele	Yannick
Anna	Stephen	Neele	Yannic
Shirley	Jacob	Clara	Tom
Amy	Frank	Klara	Nico
Angela	Larry	Leni	Niko
Virginia	Jonathan	Alina	Jacob
Brenda	Scott	Charlotte	Jakob
Catherine	Justin	Julia	Eric
Pamela	Raymond	Lisa	Erik
Katherine	Brandon	Zoe	Linus
Christine	Gregory	Zoé	Florian
Nicole	Samuel	Pia	Lennard
Janet	Patrick	Paula	Lennart
Debra	Benjamin	Melina	Nils
Carolyn	Jack	Finja	Niels
Rachel	Dennis	Finnja	Henri
Samantha	Jerry	Ida	Henry
Heather	Alexander	Lia	Nick

Tabelle A.1: Vornamen, die im Experimentkorpus als Zielakrosticha verwendet werden.

Rang	Substantive	Verben	Adjektive	Präpositionen
1	time	be	good	to
2	person	have	new	of
3	year	do	first	in
4	way	say	last	for
5	day	get	long	on
6	thing	make	great	with
7	man	go	little	at
8	world	know	own	by
9	life	take	other	from
10	hand	see	old	up
11	part	come	right	about
12	child	think	big	into
13	eye	look	high	over
14	woman	want	different	after
15	place	give	small	beneath
16	work	use	large	under
17	week	find	next	above
18	case	tell	early	
19	point	ask	young	
20	government	work	important	
21	company	seem	few	
22	number	feel	public	
23	group	try	bad	
24	problem	leave	same	
25	fact	call	able	

Tabelle A.2: Die häufigsten englischen Substantive, Verben, Adjektive und Präpositionen, die im Experimentkorpus als Zielakrosticha verwendeten.

Rang	Wort	Rang	Wort	Rang	Wort	Rang	Wort
1	the	26	they	51	when	76	*come*
2	*be*	27	we	52	*make*	77	its
3	*to*	28	*say*	53	can	78	*over*
4	*of*	29	her	54	like	79	*think*
5	and	30	she	55	*time*	80	also
6	a	31	or	56	no	81	back
7	*in*	32	an	57	just	82	*after*
8	that	33	will	58	him	83	*use*
9	*have*	34	my	59	*know*	84	two
10	I	35	one	60	*take*	85	how
11	it	36	all	61	people	86	our
12	*for*	37	would	62	*into*	87	*work*
13	not	38	there	63	*year*	88	*first*
14	*on*	39	their	64	your	89	well
15	*with*	40	what	65	*good*	90	*way*
16	he	41	so	66	some	91	even
17	as	42	*up*	67	could	92	*new*
18	you	43	out	68	them	93	*want*
19	*do*	44	if	69	*see*	94	because
20	*at*	45	*about*	70	*other*	95	any
21	this	46	who	71	than	96	these
22	but	47	*get*	72	then	97	*give*
23	his	48	which	73	now	98	*day*
24	*by*	49	*go*	74	*look*	99	most
25	*from*	50	me	75	only	100	us

Tabelle A.3: Die 100 häufigsten englischen Wörter, die im Experimentkorpus als Zielakrosticha verwendet werden. Die *markierten* Wörter kommen bereits in den Listen der Wortarten vor und werden ignoriert.