Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Ralf von der Reith

Analyse und Implementation von Such- und -Optimierungsverfahren für die Suche von Orten

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Ralf von der Reith

Analyse und Implementation von Such- und -Optimierungsverfahren für die Suche von Orten

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. rer . nat. Christoph Klauck
Zweitgutachter: Prof. Dr. Stefan Sarstedt

Eingereicht am: 12. Oktober 2018

Ralf von der Reith

Thema der Arbeit

Analyse und Implementation von Such- und -Optimierungsverfahren für die Suche von Orten

Stichworte

Algorithmus, Autovervollständigung, unscharfe Suche

Kurzzusammenfassung

Nutzer eines Informationssystems wollen mit möglichst geringem Zeitaufwand die für sie relevanten Antworten erhalten. Die interaktive, fehlertolerante Vervollständigung von Nutzeranfragen unterstützt den Nutzer bei der Eingabe und hilft ihm bei der Suche auch wenn die nötigen Suchterme nicht in Gänze bekannt sind. Diese Bachelorarbeit beschäftigt sich mit der fehlertoleranten Autovervollständigung im Rahmen eines Fahrgastinformationssystems, um den Nutzer bei der Suche nach Start- und Zielorten für eine Routenplanung zu unterstützen.

Ralf von der Reith

Title of the paper

Analysis and Implementation of search and optimization techniques for location searches

Keywords

algorithm, auto completion, approximate matching

Abstract

User of an information system expect to receive relevant feedback on their queries within minimal time. Interactive, error-tolerant completion of queries supports the user at entering the query itself as well as to find the desired data, even if the necessary search terms are unknown. This thesis deals with error-tolerant Autocompletion within a passenger-information-system, in order to help the User select the start and destination on route planning.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Motivation . . . 2
1.2 Ziele . . . 3
1.3 Aufbau der Arbeit . . . 3

2 Grundlagen **4**
2.1 Ähnlichkeit von Worten . . . 4
2.1.1 Levenshtein-Distanz . . . 4
2.1.2 Phonetische Ähnlichkeit . . . 5
2.2 Trie . . . 5
2.3 n-Gramme . . . 6
2.4 Nachbarschaftsgenerierung . . . 7
2.4.1 Löschungs-Nachbarschaft . . . 8
2.5 Verwandte Arbeiten und Abgrenzung . . . 9

3 Problemanalyse **11**
3.1 Verfügbare Daten . . . 11
3.2 Anforderungen . . . 12

4 Algorithmen und Lösungsverfahren **14**
4.1 Kanonisierung der Datensätze und Suchanfragen . . . 14
4.2 Indizierung und Vervollständigung einzelner Suchterme . . . 15
4.2.1 Exakte Autovervollständigung mit Hilfe eines Tries . . . 15
4.2.2 ICAN - Imcrementally Compute Active Nodes . . . 18
4.2.3 ICPAN - Imcrementally Compute Pivotal Active Nodes . . . 22
4.2.4 IncNGTree - Incremental algorithm based on Neighborhood Generation and a Trie index . . . 26
4.2.5 META - Matching-based framework for Error-Tolerant Autocompletion 28
4.3 Bilden der Schnittmenge . . . 30
4.4 Priorisierung der Kandidaten . . . 31

5 Konzeption und Implementierung der Suchmaschine **33**

6 Tests und Evaluation **39**
6.1 Tests . . . 39
6.2 Evaluation . . . 40
6.2.1 Analyse der META-Suchmaschine . . . 40

6.2.2 Vergleich von NGram- und META-Suchmaschine 43

7 Zusammenfassung und Ausblick 46
7.1 Zusammenfassung 46
7.2 Ausblick 46

Tabellenverzeichnis

2.1 Berechnung der Editierdistanz zwischen den Wörtern „Bahnhof“ und „Baum“ 4

6.1 Laufzeit der Indexerstellung für den HVV-Datensatz 44

Abbildungsverzeichnis

2.1 Ein Trie bestehend aus den Worten „Altona“, „Alter“, „Wall“, „Horn“ und „HBF“ 6

4.1 Exakte Suche des Terms „alt“ in einem Trie 17
4.2 Suche des Terms „alt“ in einem Trie mit Hilfe des ICAN-Algorithmus bei maximaler Editier-Distanz $\tau = 2$ 19
4.3 Suche des Terms „alt“ in einem Trie mit Hilfe des IPCAN-Algorithmus bei maximaler Editier-Distanz $\tau = 2$ 25
4.4 IncNGTrie der Worte „Horn“ und „HBF“ mit einer Editierdistanz von $\tau = 1$ . . 27

5.1 UML-Klassendiagramm: METASuchmaschine 35
5.2 UML-Sequenzdiagramm: Initialisierungsphase der METASuchmaschine 36
5.3 UML-Sequenzdiagramm: Bearbeitung einer Suchanfrage an die METASearchEngine 38

6.1 Indexerstellung - DLBP Laufzeit 41
6.2 Suchanfrage - Laufzeit, Indexgröße variabel 42
6.3 Suchanfrage - Laufzeit, Suchtermlänge variabel 42
6.4 Suchanfrage - Laufzeit, Anzahl Suchterme variabel 43
6.5 Suchanfrage - Laufzeit auf dem HVV-Datensatz 45

1 Einleitung

In vielen Informationssystemen müssen die Benutzer bisher vollständige Suchanfragen mit exakten Schlüsselwörtern und frei von Rechtschreibfehlern formulieren, um die gewünschten Daten zu erhalten. Häufig kennen die Nutzer jedoch nicht die genaue Struktur der Daten und mit zunehmender Länge der Suchanfrage schleichen sich Rechtschreibfehler ein. Durch die fehlertolerante Suche und Vorschläge zur Autovervollständigung lassen sich die Auswirkungen dieser Problematiken reduzieren. Autovervollständigung ist ein interaktiver Prozess, bei dem Vorschläge generiert und angezeigt werden um die Eingabe fortzusetzen, während der Benutzer sie um weitere Zeichen ergänzt. Sie findet heutzutage in vielen verschiedenen Bereichen Anwendung. In Entwicklungsumgebungen und Terminals werden sie genutzt, um den Nutzer bei der schnelleren Eingabe von Befehlen zu unterstützen. In Suchmaschinen unterstützt es die Nutzer nicht nur in der schnelleren Eingabe sondern bietet durch die Vervollständigung ganzer Phrasen auch eine weitere Möglichkeit die durchsuchbaren Daten zu erkunden und wurde in diesem Kontext zuerst durch Google Suggest im Jahr 2004 (Garber (2013)) eingebunden. Auch auf mobilen Endgeräten findet Autocomplete zunehmend Verwendung, da die Eingabe von Text auf diesen Geräten oftmals mühsam und fehleranfällig ist.

Während die exakte Autovervollständigung vorallem die Anzahl der einzugebenden Zeichen und damit die Eingabezeit reduziert, beschäftigt sich die Forschung zuletzt auch mit der Erweiterung um Fehlertoleranz, um den Einfluss von Rechtschreibfehlern oder Unwissen über die korrekte Schreibweise zu reduzieren. Diese Forschungen sind insbesondere für den Bereich des *Information Retrieval* von Bedeutung, in dem Rechtschreibfehler oftmals nicht nur in der Suchanfrage sondern auch in den zu durchsuchenden Dokumenten enthalten sind (Celikik und Bast (2009)) und so das Finden von relevanten Ergebnissen oftmals zusätzlich erschwert wird.

Die unscharfe Suche und die Autovervollständigung von Eingaben stellen nach Chaudhuri und Kaushik (2009) zunächst einmal zwei unabhängige Dimensionen dar, welche beide von der Wissenschaft gut untersucht sind. Die Kombinierung dieser beiden Dimensionen ist jedoch eine Herausforderung, die in der Forschung bisher vergleichsweise wenig Aufmerksamkeit erhalten hat.

1.1 Motivation

Fahrplanauskunftsysteme unterstützen den Benutzer in der schnellen und unkomplizierten Planung von Routen. Ein wichtiger Bestandteil dieser Systeme ist die Suche und Auswahl von Orten, die sich als Start- und Zielpunkt für die anschließende Routenplanung verwenden lassen. Wie auch im E-Commerce bei der Suche nach Produkten wollen die Nutzer schnell die gewünschten Informationen finden und möglichst wenig Zeit auf die Suche verwenden. Autovervollständigungsfunktionen sind ein effizienter Ansatz, die Eingabezeit durch die Präsentation relevanter Vorschläge zu verringern. Hinzu kommt, dass Nutzer sich insbesondere bei längeren oder komplizierten Namen, wie das bei Straßen zum Teil der Fall sein kann, häufiger verschreiben oder die genaue Schreibweise nicht bekannt ist. Autovervollständigung unterstützt den Nutzer dabei, solcherlei Fehler zu vermeiden, indem der Nutzer durch die Anzeige von Vorschlägen in seiner Eingabe gelenkt wird und bereits ein Teil der Eingabe ausreicht, um den gewünschten Eintrag zu finden. Allerdings reicht bereits ein Schreibfehler aus, sodass der Eintrag auch mit Hilfe der Vorschläge nicht mehr gefunden wird.

Aus diesem Grund werden wir eine Autovervollständigung mit unscharfer Suche vereinen, um auch eine geringe Menge an Fehlern in der Eingabe zu tolerieren. Aufgrund der interaktiven Natur von Auto-Vervollständigungsanwendungen erwarten Nutzer eine sofortige Rückmeldung des Systems. Die vorherige Arbeit von Miller (1968) zeigt, dass eine maximale Reaktionszeit des Systems von bis zu 100ms angemessen ist. Dies schließt allerdings auch die Netzwerk-Kommunikation zwischen Server und Client und anderweitige Einflussfaktoren mit ein. Deshalb sollte die Rechenzeit der Suche minimiert werden.

Eine besondere Herausforderung ist dabei, die große Menge an möglichen Kandidaten in dieser Zeit zu berechnen und diese auf eine eher kleine Menge der relevantesten Kandidaten zu reduzieren. Mit steigender Größe der zugrunde liegenden Datenbasis wird es immer schwerer diese Anforderung zu erfüllen, da die Anzahl der Kandidaten, die berücksichtigt und am Ende nach ihrer Relevanz bewertet werden müssen, steigt. Insbesondere durch die unscharfe Suche wird der potenzielle Recall vor allem für kürzere Suchanfragen noch einmal deutlich erhöht.

Ab einer gewissen Größe des zugrundeliegenden Datensatzes sind diese Herausforderungen mit simplen, linearen Suchverfahren oft nicht mehr zu bewältigen, sodass die zu betrachtenden Datensätze durch die Auswahl geeigneterer Indexstrukturen und Algorithmen beschleunigt werden muss.

1.2 Ziele

Ziel dieser Arbeit ist es, verschiedene Verfahren zur fehlertoleranten Autovervollständigung für die Suche nach Orten in dem Fahrplanauskunftsystem Geofox zu testen. Das Hauptaugenmerk liegt dabei weniger auf der Entwicklung eines hochoptimierten Verfahrens, sondern darauf, verschiedene Ansätze zu analysieren, zu vergleichen und auf ihre grundsätzliche Eignung zu prüfen.

Dazu werden in einem ersten Schritt die detaillierten Anforderungen aus Gesprächen mit dem Betreiber, sowie bereits existierenden Dokumenten extrahiert. Anschließend werden einige Verfahren aus der Wissenschaft betrachtet und auf die theoretische Eignung für unseren Anwendungsfall bewertet. Im Anschluss daran wird das Konzept für einen Prototypen entwickelt und dieser implementiert. Abschließend folgt eine Untersuchung der Performance des Prototypen, sowie ein Vergleich mit der aktuell im Einsatz befindlichen Suchmaschine.

1.3 Aufbau der Arbeit

Diese Arbeit ist in 6 Abschnitte unterteilt. Kapitel 2 beschäftigt sich mit einigen Begriffsdefinitionen und Datenstrukturen, auf die die später vorgestellten Algorithmen aufbauen. In Kapitel 3 werden zum einen die zur Verfügung stehenden Daten und zum Anderen die Anforderungen an eine Suchmaschine im Kontext der Geofox-Anwendung beschrieben. Kapitel 4 widmet sich der aktuellen Forschung in diesem Themenfeld und stellt einige relevante Algorithmen vor. Darauf folgt in Kapitel 5 die Bewertung der Algorithmen gefolgt von Konzeption und Implementation der Suchmaschine. Das Kapitel 6 umfasst die Beschreibung des Testrahmens für die aktuelle und die zu entwickelnde Suchmaschine und präsentiert die Ergebnisse.

2 Grundlagen

2.1 Ähnlichkeit von Worten

Da wir beabsichtigen, eine unscharfe Suche für die Autovervollständigung zu implementieren, benötigen wir eine Möglichkeit, ähnliche Worte zu finden. Eines der wohl bekanntesten und in der Wissenschaft gut untersuchten Verfahren stellt die Levenshtein-Distanz dar.

2.1.1 Levenshtein-Distanz

Bei der Levenshtein-Distanz (oft auch einfach „Editierdistanz" genannt) handelt es sich um eine Metrik, die die Ähnlichkeit von zwei Zeichenketten darauf basierend bestimmt, wie viele Zeichen in einer Zeichenfolge s_1 eingefügt, gelöscht oder substituiert werden müssen, um s_1 in eine andere Zeichenfolge s_2 zu überführen (Navarro u. a. (2000)). Je geringer die Anzahl der Modifikationen, desto ähnlicher sind sich die zwei Zeichenfolgen.

Die Levenshtein-Distanz von zwei Zeichenfolgen s_1 und s_2 kann mithilfe eines gut untersuchten Ansatzes aus der Dynamischen Programmierung berechnet werden. Dabei werden die zwei Zeichenfolgen in eine Matrix eingefügt, s_1 von oben nach unten, s_2 von links nach rechts. Anschließend wird die Editierdistanz inkrementell von allen Präfixen der beiden Wörter berechnet. Die Komplexität dieses Verfahrens beträgt somit $\mathcal{O}(|s_1| * |s_2|)$, wobei $|s|$ die Länge eines Wortes bezeichnet. Tabelle 2.1 zeigt eine solche Matrix für die beiden Wörter „Bahnhof" und „Baum".

#		**B**	**A**	**H**	**N**	**H**	**O**	**F**
	0	1	2	3	4	5	6	7
B	1	0	1	2	3	4	5	6
A	2	1	0	1	2	3	4	5
U	3	2	1	1	2	3	4	5
M	4	3	2	2	2	3	4	5

Tabelle 2.1: Berechnung der Editierdistanz zwischen den Wörtern „Bahnhof" und „Baum"

Es gibt verschiedene Vereinfachungen und Optimierungen für dieses Verfahren, welche die Laufzeit zum Teil erheblich verbessern. Außerdem gibt es häufig verwendete Erweiterungen wie z.B. die Damerau-Levenshtein Distanz, welche auch Transpositionen, also das Vertauschen von Buchstaben als eine einzelne Modifikation betrachtet, da dies ein häufiger Fehler bei der Eingabe ist. Diese Verfahren werden, ebenso wie weitere Metriken zur Berechnung von Distanzen zwischen zwei Zeichenfolgen, hier allerdings nicht näher erläutert, da sie in dieser Arbeit keine Verwendung finden. In den hier vorgestellten Verfahren erfolgt die Berechnung der Levenshtein-Distanz meist implizit während der Traversierung eines Tries, einer Datenstruktur zur Speicherung von Zeichenfolgen, die später noch näher erläutert wird.

2.1.2 Phonetische Ähnlichkeit

Neben der oben erwähnten Levenshtein-Distanz und anderen Metriken gibt es auch phonetische Verfahren zur Bestimmung der Ähnlichkeit von Worten. Der wohl bekannteste Vertreter ist Soundex, der vorallem für die Englische Sprache genutzt wird. Ein bekanntes phonetisches Verfahren für die Deutsche Sprache wäre die Kölner Phonetik. Bei Soundex wie auch der Kölner Phonetik wird allen Wörtern ein Code basierend auf dem Sprachklang zugeordnet, sodass Wörter mit gleichem Klang einen identischen Code erhalten. So erzeugen in der Kölner Phonetik „Meier“, „Maier“ als auch „Meyer“ alle den Code 67.

2.2 Trie

Ein Trie (abgeleitet von Information Re*trie*val (Knuth, 1973, S.481)), ist eine baumartige Datenstruktur zur Speicherung und Suche von Zeichenketten. Jeder Knoten ist mit einem Zeichen aus dem Alphabet Σ versehen, sodass jeder Pfad von der Wurzel zu einem Blatt eine im Trie gespeicherte Zeichenkette darstellt. Die Suche nach einer Zeichenkette s folgt nun dem Prinzip des Daumenregisters von Wörterbüchern. Beginnend bei der Wurzel wird der Baum entsprechend der Zeichenfolge aus s durchlaufen. Ist ein entsprechender Kindknoten nicht vorhanden oder endet die Zeichenfolge nicht auf einem als Wortende markierten Knoten, ist diese Zeichenfolge nicht im Trie gespeichert. Abbildung 2.1 zeigt einen Trie bestehend aus den Worten „Altona“, „Alter“, „Wall“, „HBF“ und „Horn“.

Diese Datenstruktur bringt einige Vorteile mit sich. Zum Einen sind neben den Wörtern selbst auch sämtliche Präfixe gespeichert, welche in einem Autovervollständigungs-Szenario von Interesse sein können. Wenn sich mehrere Wörter ein Präfix teilen, wird auch eine gewisse Datenkompression erreicht, da jedes Präfix nur einmal gespeichert wird. Außerdem ist die Suche nach Wörtern unabhängig von der Größe des Index und vergleichsweise schnell mit

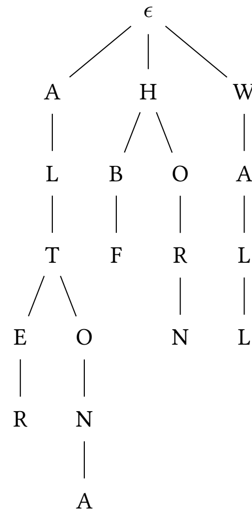


Abbildung 2.1: Ein Trie bestehend aus den Worten „Altona“, „Alter“, „Wall“, „Horn“ und „HBF“

einer durchschnittlichen Laufzeit von $\mathcal{O}(l)$ mit l als durchschnittliche Länge der gespeicherten Zeichenfolgen.

Eine naive Implementation des Trie besteht darin, in jedem Knoten zu jedem Zeichen aus dem verwendeten Alphabet eine Referenz auf potentielle Folgeknoten zu speichern. Während dies die effizienteste Implementierung für einen schnellen Zugriff ist, verschwendet sie viel Speicherplatz, da insbesondere in den unteren Ebenen des Tries nur zu wenigen Symbolen aus dem Alphabet noch Folgeknoten existieren und somit viel Speicherplatz für Null-Referenzen reserviert wird. Nun gibt es verschiedene Techniken, den Speicherbedarf auf Kosten der Laufzeit zu reduzieren. Ein Ansatz ist, stattdessen die Kindknoten nach ihrer Beschriftung sortiert in einer Liste zu speichern, um so die Binärsuche nach einem gewünschten Nachfolger zu ermöglichen.

2.3 n-Gramme

Als n-Gramme werden Teilfolgen der Länge n einer Zeichenfolge bezeichnet. Sie werden genutzt, um die Menge der zu betrachtenden Wörter des Wörterbuchs zu filtern und (meistens) das Problem der unscharfen Suche auf die exakte Suche von Teilfolgen zu reduzieren. Es gibt viele verschiedene Ansätze, wie n-Gramme eingesetzt werden können. Einer der einfacheren Ansätze ist es, alle Wörter des Wörterbuches in ihre n-Gramme zu zerlegen und diese als invertierte Listen zu indizieren. Diese invertierten Listen speichern zu jedem n-Gramm alle

Wörter, in denen dieses n-Gramm enthalten ist. Jede Zeichenfolge s lässt sich dabei in $|s|-n+1$ n-Gramme zerlegen.

Beispiel 1 *Bilden wir für das Wort „Bahnhof“ alle dazugehörigen 3-Gramme bzw Trigramme, erhalten wir* $|s_{Bahnhof}| = 7$; $|s_{Bahnhof}| - n - 1 = 7 - 3 + 1 = 5$ *n-Gramme.*

$$Bahnhof \rightarrow [Bah, ahn, hnh, nho, hof]$$

Wird nun nach einem Wort s gesucht, werden für q ebenfalls die n-Gramme erzeugt und diese anschließend in dem n-Gramm-Index gesucht. Für jedes der n-Gramme erhalten wir nun eine Liste an Einträgen. Jeder Eintrag, der eine ausreichende Menge an zugehörigen n-Grammen aus unserer Suchanfrage aufweist, ist ein Kandidat für unsere Suchanfrage (Boytsov (2011)).

2.4 Nachbarschaftsgenerierung

Nachbarschaftsgenerierung ist ein Ansatz, der das Problem der unscharfen Suche von Zeichenketten auf die exakte Suche von Zeichenketten herunterbricht. Um dies zu erreichen, werden alle Zeichenketten q_n erzeugt, die *benachbart* zu der Suchanfrage q sind und anschließend für jede dieser generierten Zeichenkette eine exakte Suche in unserem Wörterbuch durchgeführt. Nachbarschaft ist dabei wie folgt definiert:

Defintion 1 (Nachbarschaft) *Zwei Zeichenfolgen s_1 und s_2 werden als* benachbart *bezeichnet, wenn s_1 innerhalb einer festgelegten, maximalen Editierdistanz τ zu s_2 liegt.*

Entsprechend bezeichnet der Begriff der *k-Nachbarschaft* eine Nachbarschaft mit maximaler Editierdistanz $\tau = k$. Zur Veranschaulichung folgt die 1-Nachbarschaft $U_1(q)$ von einer Zeichenfolge $q = abc$ auf Basis eines Alphabets Σ aller kleinen Buchstaben $\Sigma = [a - z]$:

- abc selbst ist enthalten.
- Alle Zeichenfolgen mit einer Löschung: ab, ac, bc
- Alle Zeichenfolgen mit einer Substitution:
  $bbc, cbc, \ldots, zbc; aac, acc, adc, \ldots, azc; aba, \ldots, abz$
- Alle Zeichenfolgen mit Einfügung eines Buchstabens:
  $aabc, babc, \ldots, zabc; aabc, \ldots, azbc; abac, \ldots, abzc; abca, \ldots, abcz$

Daraus ergibt sich allein für eine dreistellige Zeichenfolge eine Nachbarschaft mit $1 + 3 + 3 * 25 + 4 * 26 = 183$ Elementen. Davon sind 180 Elemente einzigartig. Das Problem einer vollständigen k-Nachbarschaft ist somit der exponentielle Anstieg dieser generierten Menge an Zeichenketten mit $|U_k(q)| = \mathcal{O}(|q|^k * |\Sigma|^k)$ (Boytsov (2011)). Dadurch ist die vollständige k-Nachbarschaft nur für sehr kleine Alphabete und Werte von k sinnvoll einsetzbar.

Neben der vollständigen Nachbarschaft gibt es auch die sogenannte *Löschungs-Nachbarschaft*, welche eine Teilmenge der vollständigen Nachbarschaft darstellt.

2.4.1 Löschungs-Nachbarschaft

Bei der Löschungs-Nachbarschaft werden nur all jene Nachbarn erzeugt, die ausschließlich durch Lösch-Operationen erzeugt werden können. Die Löschungs-Nachbarschaft $U_k^d(s)$ von $s_q = abc$ wäre also $U_1^d(s_q) = abc, ab, ac, bc$.

Das Problem der unscharfen Suche kann nun auf Basis der Löschungs-Nachbarschaft mit der Mor-Fraenkel Methode (siehe (Boytsov, 2011, 6.2.7)) gelöst werden, welche in einem später vorgestellten Verfahren eine Rolle spielt. Die Mor-Fraenkel Methode basiert auf der Beobachtung, dass wenn zwei Zeichenfolgen s_1 und s_2 innerhalb der Editierdistanz k zueinander liegen, es eine Zeichenfolge v gibt, die aus s_1 und s_2 durch je $l \leq k$ Löschungen generiert werden kann. Ein einfaches Verfahren, welches die Löschungsnachbarschaft zum finden ähnlicher Wörter nutzt, generiert für jeden Datensatz die Löschungs-Nachbarschaft und indiziert die resultierenden Zeichenfolgen. Wird nun eine Suche gestartet wird auch die Löschungs-Nachbarschaft für die Suchanfrage generiert, und für jede der resultierenden Zeichenfolgen, wie auch bei der vollständigen Nachbarschaft, eine exakte Suche initiiert. Wie das folgende Beispiel zeigt ist dieses Vorgehen jedoch anfällig für false-positives.

Beispiel 2 *Sei $s = alen$ und $q = lena$, sowie $\tau = 1$, dann kann sowohl aus s, als auch aus q mit einer Löschung $q_{[1-3]} = s_{[2-4]} = len$ generiert werden, obwohl die Editier-Distanz zwischen s und q $k = 2$ beträgt.*

Um diese false-positives zu erkennen, werden in der Mor-Fraenkel Methode zusätzlich die Positionen und Buchstaben der Löschungen einer Zeichenfolge gespeichert:

Zuerst wird die Löschungs-Nachbarschaft aller Wörter s des Wörterbuchs erzeugt und Tripel der Form (s', D^s, C^s) unter dem Schlüsselwort s' indiziert, wobei s' eine Zeichenfolge ist, die durch $l \leq k$ Löschungen in s entstanden ist. D^s ist eine Liste mit den Positionen aller gelöschten Buchstaben und C^s ist eine Liste, die die gelöschten Buchstaben enthält. Die Reihenfolge der Elemente in D^s und C^s entspricht der Reihenfolge der Löschungen aus s.

Wenn nun eine Suchanfrage p gestellt wird, werden auch für p alle Tripel (p', D^p, C^p) der Löschungs-Nachbarschaft generiert und es wird nach allen p' im Index gesucht. Anschließend werden alle Tripel (s', D^s, C^s) zurückgegeben, die die folgenden Bedingungen erfüllen:

$$p' = s'$$
$$|D^s| + |D^p| - |D^s \cap D^p| \leq k$$

$|D^p| - |D^s \cap D^p|$ spiegelt dabei die Anzahl der Löschungen, $|D^s| - |D^s \cap D^p|$ die Anzahl der Einfügeoperationen und $|D^s \cap D^p|$ die Anzahl der nötigen Substitutionen wider. Aus der Zeichenfolge s', sowie den Listen D^s und C^s kann nun die ursprüngliche Zeichenfolge wiederhergestellt werden. Den jeweiligen Zeichenfolgen s' können auch Assoziationen zu anderen Objekten oder Dokumenten hinzugefügt werden, indem das Tripel um eine entsprechende Referenz erweitert wird. Das Problem bei der Löschungs-Nachbarschaft ist die Größe des entstehenden Index, der für die Englische Sprache grob 140 mal so groß ist, wie das ursprüngliche Wörterbuch (siehe (Boytsov, 2011, Table IX)).

2.5 Verwandte Arbeiten und Abgrenzung

Die wissenschaftlichen Bereiche die dieser Arbeit am nächsten kommen lassen sich grob in die Bereiche der fehlertoleranten Autovervollständigung (Error Tolerant Autocompletion - ETA) oder auch dem Themenfeld der string similarity search und der Anfragen-Autovervollständigung (Query Autocompletion - QAC) unterteilen.

ETA beschäftigt sich vorallem damit, möglichst performante Algorithmen zum Durchsuchen großer Datenbestände auf für eine Anfrage relevante Datensätze zu entwickeln. Ziel ist dabei auch, einen vollständigen Recall zu erhalten, sodass potentiell relevante Ergebnisse auch wirklich enthalten sind.

QAC hingegen beschäftigt sich vorallem mit der Priorisierung des Recalls, um so nur die für den Nutzer relevantesten Einträge bereitzustellen. Hier wird vorallem mit Log-Daten und auch anderen Informationen gearbeitet, um z.B. über Trends und Beliebtheit bestimmter Einträge oder auch andere Metainformationen wie z.B. der Einbeziehung von Interessen des Nutzers eine möglichst relevante, aber möglicherweise nicht vollständige Liste von Ergebnissen zu präsentieren.

Aus diesem Grund ist für uns das Feld des QAC weniger von Bedeutung, da Nutzer unseres Fahrgastsystems auch abgelegenere Stationen schnell finden können sollen und die Priorisie-

rung vorallem auf Basis der Editierdistanz und auf statischen Daten wie Stationstypen basieren werden.

3 Problemanalyse

Nutzer eines Fahrplanauskunftsystems wollen mit minimalem Aufwand eine präzise Auskunft über die gewünschten Routen erhalten. Ein Teil des Prozesses ist die Eingabe und Auswahl der Start- und Zielorte. Vorschläge zur Autovervollständigung einer begonnenen Eingabe des Nutzers tragen erheblich dazu bei, die Eingabezeit bis zur Auswahl eines Eintrages zu reduzieren und Eingabefehler zu vermeiden. Im Folgenden werden die Anforderungen an die Autovervollständigung sowie die dafür zur Verfügung stehenden Daten im Detail erläutert.

3.1 Verfügbare Daten

Bei den zur Verfügung stehenden Daten handelt es sich zum einen um Informationen von Start- und Zielorten und zum anderen um Metadaten, wie zum Beispiel Logs über bisherige Suchanfragen, aus denen sich bspw. die Häufigkeit ermitteln lässt, wie oft ein Ort gesucht wird.

Um spezifische Start- und Zielorte und die Orte im Sinne von Städten/Gemeinden/etc. klarer voneinander abzugrenzen folgen nun zwei Definitionen, welche für den weiteren Verlauf dieser Arbeit gelten:

Lokationen bezeichnen im Folgenden alle spezifischen Orte, die als Start- oder Zielort der Routenplanung verwendet werden können.

Orte bezeichnen im Folgenden geografische Gebiete wie Städte, Dörfer, Gemeinden o.Ä. in denen sich eine Lokation befindet.

Es gibt mehrere Arten von Lokationen, welche als Start- und Zielort ausgewählt werden können:

Addressen setzen sich aus Name, Ort und Hausnummer zusammen.

Stationen setzen sich aus dem Namen der Station, Ort und Gauß-Krüger Koordinaten zusammen.

Points of Interest/Sehenswürdigkeiten setzen sich aus Name, Ort, Adresse, Kategorie, Webseite und Gauß-Krüger Koordinaten zusammen.

Alle Datensätze stammen aus dem HVV-Gesamtbereich und Teilen der umliegenden Bundesländer. Insgesamt sind etwa 60.000 Einträge unter 100.000 Namen enthalten. Ergänzend zu den oben genannten Lokationen gibt es für viele Haltestellen und POIs auch Kürzel und Alias-Namen, über die diese gefunden werden können. Außerdem gibt es Überschneidungen bei den Namen von Lokationen verschiedener Art. So existieren beispielsweise eine Straße, eine Sehenswürdigkeit und auch eine Station namens Reeperbahn in Hamburg. Darüber hinaus sind Orte hierarchisch gegliedert (z.B. ist Maschen ein Ort in der Gemeinde Seevetal. Seevetal ist nach obiger Definition ebenfalls ein Ort).

3.2 Anforderungen

Der Zweck von Autovervollständigungs-Funktionen ist es, die Anzahl der eingegebenen Zeichen und damit die Eingabezeit und das Fehlerpotenzial zu reduzieren, da die Eingabe insbesondere bei Mobilgeräten oft mühsam und fehleranfällig ist. Eine wichtige Eigenschaft der Autovervollständigung ist ihre Interaktivität - Vorschläge werden ohne Verzögerung angezeigt, während der Nutzer tippt. Vorherige Arbeiten (Miller (1968)) zeigen, dass eine maximale Reaktionszeit des Systems von bis zu 100ms angemessen ist. Um den Nutzer bei der Eingabe zusätzlich zu unterstützen, soll die Eingabe auch tolerant gegenüber Eingabefehlern sein. Dies unterstützt den Nutzer nicht nur bei Rechtschreibfehlern, sondern auch wenn ihm die genaue Schreibweise einer Lokation nicht bekannt ist. Da den Nutzern die interne Struktur der Datensätze nicht bekannt ist, darf die Reihenfolge der Schlüsselwörter keinen Einfluss auf die Auswahl der Vorschläge haben (z.B. Straßenname, Stadtname ↔ Stadtname, Straßenname). Um die Eingabe zusätzlich zu vereinfachen wird Groß- und Kleinschreibung nicht beachtet. Leerzeichen und Sonderzeichen trennen mehrere Schlüsselwörter von einander und werden ansonsten ebenfalls ignoriert. Neben der Reaktionsgeschwindigkeit ist aber auch die Qualität der zurückgegebenen Einträge von hoher Bedeutung. Gerade bei kürzeren Eingaben ist die Menge der zurückgegebenen Vorschläge oft erheblich. Durch die zusätzliche Toleranz von Fehlern wird die Anzahl möglicher Treffer noch einmal deutlich erweitert. Darüber hinaus soll es die Möglichkeit geben, bestimmte Einträge gegenüber anderen zu priorisieren. Für Geofox bedeutet das, dass Einträge aus Hamburg eine höhere Gewichtung erhalten, als Einträge aus dem Umland und Bahnhöfe gegenüber anderen Lokationen bevorzugt werden. Darüber hinaus wird bei kombinierten Haltestellen und Stationen mit identischem Namen jeweils nur ein bestimmter Eintrag angezeigt. Werden Anfragen mit Ziffern gestellt, gehen wir davon aus, dass Addressen gesucht werden. Dementsprechend werden in diesem Fall Addressen, ansonsten jedoch Stationen und Sehenswürdigkeiten höher priorisiert.

Der Übersichtlichkeit und späteren Referenzierbarkeit halber folgt eine Auflistung der obigen Anforderungen.

A1 Aus dem Datenbestand sollen Vorschläge zur Vervollständigung einer begonnenen Eingabe generiert werden.

A2 Die Vervollständigung ist tolerant gegenüber Schreibfehlern.

A2.1 Groß- und Kleinschreibung wird nicht beachtet.

A2.2 Leer- und Sonderzeichen[1] trennen einzelne Schlüsselwörter voneinander und werden ansonsten ignoriert.

A2.3 Die Reihenfolge der Schlüsselwörter hat keinen Einfluss auf die Auswahl der Suchvorschläge.

A3 Die Antwortzeit des Systems darf 100ms nicht überschreiten.

A4 Die Datensätze können gewichtet werden. Diese Gewichtung wirkt sich auf die Auswahl und Reihenfolge der anzuzeigenden Vorschläge aus.

A4.1 Lokationen in Hamburg werden gegenüber Stationen außerhalb Hamburgs höher priorisiert.

A4.2 Bahnhöfe werden höher priorisiert als andere Lokationen.

A5 Bei gleichnamigen Einträgen wird nur eine Auswahl der Einträge angezeigt.

A5.1 Bei kombinierten Bus- und Schnellbahn- Haltestellen wird nur die Schnellbahn-Haltestelle angezeigt.

A5.2 Enthält die Suchanfrage Ziffern, werden bei gleichnamigen Lokationen verschiedener Art Straßen gegenüber POIs und Stationen bevorzugt. Ist keine Ziffer in der Suchanfrage enthalten, werden POIs und Stationen gegenüber Straßen bevorzugt. Die bevorzugten Einträge werden höher priorisiert.

[1] Als Sonderzeichen gelten alle nicht-alphanumerischen Zeichen [ˆa-zA-Z0-9]

4 Algorithmen und Lösungsverfahren

In diesem Kapitel werden verschiedene, relevante Verfahren aus der bisherigen wissenschaftlichen Arbeit in diesem Feld vorgestellt, sowie ihre Vor- und Nachteile erläutert. Grundsätzlich lässt sich die fehlertolerante Autovervollständigung in mehrere Bestandteile zerlegen: In einem ersten Schritt muss ein Index aufgebaut werden. Dazu werden alle Datensätze kanonisiert und in Schlüsselwörter zerlegt. Diese Schlüsselwörter werden dann zusammen mit einer Referenz an die ursprünglichen Datensätze indiziert. Sobald der Index konstruiert wurde, können Suchanfragen gestellt werden. Auch die Suchanfragen werden kanonisiert und in ihre Schlüsselwörter zerlegt. Anschließend wird nach jedem Schlüsselwort der Suchanfrage im Index gesucht. Für jedes dieser Schlüsselwörter wird eine möglicherweise leere Liste an Kandidaten zurückgegeben. Anschließend wird die Schnittmenge dieser Listen gebildet und zuletzt diese Ergebnisse auf Basis einer Gewichtung sortiert und als Ergebnis für die Suchanfrage zurückgegeben.

Entsprechend dieses Vorgehens ist auch dieses Kapitel strukturiert. Zuerst wird die Kanonisierung und Zerlegung der Datensätze und Suchanfragen in Schlüsselworte beschrieben. Anschließend folgt die Beschreibung der jeweiligen Suchverfahren für einzelne Schlüsselwörter mit dem dazugehörigen Index. Daraufhin folgen Verfahren zur Bildung der Schnittmenge, gefolgt von den Verfahren zur Priorisierung. Abschließend wird ein Verfahren ausgewählt, um dieses in **??** näher zu untersuchen und auf Basis des bisher im Einsatz befindlichen Verfahrens zu bewerten.

4.1 Kanonisierung der Datensätze und Suchanfragen

Sämtliche Datensätze und gestellte Suchanfragen, der Einfachheit halber im folgenden „Zeichenfolgen“ genannt, werden kanonisiert und in Schlüsselworte zerlegt. Auf Basis der Anforderung A2.1 werden alle Zeichenfolgen zuerst in Kleinbuchstaben umgewandelt. Anschließend werden alle Ziffern entfernt. Außerdem werden sämtliche Umlaute, sowie das „ß“ wie folgt ersetzt:

- ß → ss
- ä → ae

- ö → oe
- ü → ue

Anschließend wird die Zeichenfolge entsprechend der Anforderung A2.2 in ihre einzelnen Schlüsselwörter zerlegt. Dazu wird die Zeichenfolge bei jedem Vorkommen eines nicht-alphanumerischen Zeichens ([ˆa-zA-Z0-9]) in Teilfolgen getrennt. Jede daraus entstehende nicht-leere Zeichenfolge stellt nun ein Schlüsselwort dar, welches für die weitere Suche genutzt wird.

Beispiel 3 *Es wird eine Suchanfrage „Hamburg, Straßburger Straße 5“ gestellt. Durch Kanonisierung der Suchanfrage entsteht nun „hamburg, strassburger strasse 5“. Nach der Trennung in Schlüsselwörter somit die Liste [„hamburg“, „strassburger“, „strasse“].*

4.2 Indizierung und Vervollständigung einzelner Suchterme

Nahezu alle wissenschaftlichen Arbeitem im Bereich der fehlertoleranten Autovervollständigung nutzen einen Trie als Index-Struktur, da dieser bereits die Präfixe enthält und das finden von Vervollständigungen sehr einfach macht. Für das bessere Verständnis der nachfolgenden Algorithem wird zuerst die Autovervollständigung ohne Fehlertoleranz erläutert. Die hier vorgestellten Verfahren gehen von bereits kanonisierten und in Suchterme zerlegten Anfragen aus und betrachten lediglich das Suchen eines der Suchtermes.

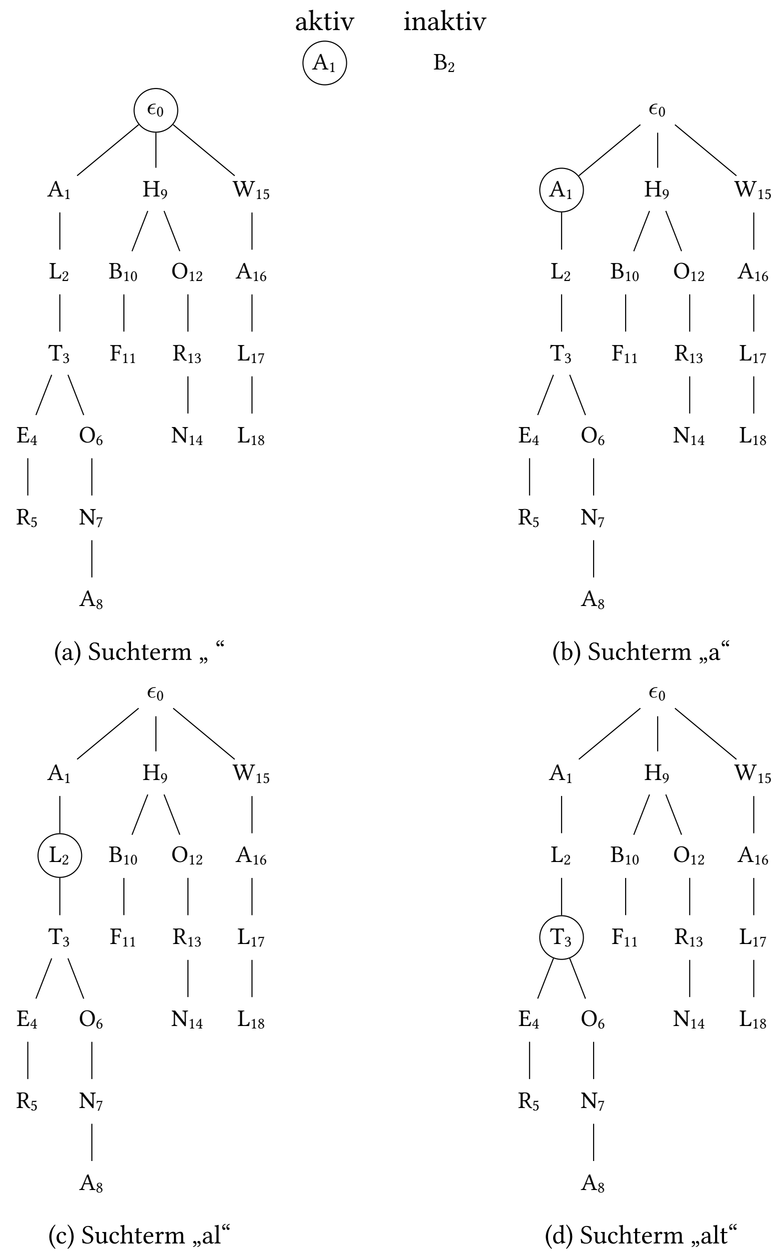
4.2.1 Exakte Autovervollständigung mit Hilfe eines Tries

In einem ersten Schritt wird der Index aufgebaut. Dafür werden alle Schlüssel kanonisiert und in seine Schlüsselwörter zerlegt. Diese Schlüsselwörter werden in einem Trie gespeichert. Die Knoten, die das Ende eines Schlüsselwortes repräsentieren enthalten eine Referenz auf die dazu korrelierenden Datensätze. Beginnt der Nutzer nun seine Eingabe $s_x = c_1c_2 \dots c_x$ wird in den Kindknoten der Wurzel nach einem Knoten mit c_1 als Beschriftung gesucht. Wird dieser gefunden, wird von diesem ausgehend nun ein Kindnoten mit c_2 gesucht. Dies wird fortgeführt, bis die Eingabezeichenkette vollständig konsumiert wurde. Um nun von dem gefundenen Knoten die Autovervollständigungen zu erhalten, werden alle Blätter, die Nachkommen dieses Knotens sind, aufgesucht und die referenzierten Datensätze extrahiert. Existiert kein Pfad für die eingegebene Zeichenkette, ist kein Wort mit diesem Präfix im Trie hinterlegt.

Bei einer naiven Vorgehensweise würde das Ergebnis für jede Anfrage von Anfang an, bei der Wurzel beginnend, neu berechnet werden. Wenn nun ein Nutzer seine Eingabe s_x mit einem

weiteren Zeichen c_{x+1} fortsetzt, würde dies zu vielen redundanten Neuberechnungen für alle Buchstaben von c_1 bis c_x führen. Wenn für jede Anfrage s_x stattdessen der entsprechende Knoten n_x gecached wird, kann von diesem ausgehend die Berechnung für c_{x+1} fortgesetzt werden indem wir lediglich prüfen, ob n_x einen entsprechenden Kindknoten besitzt.

Beispiel 4 *Wir erstellen einen Trie aus den Datensätzen „Altona“, „Alter Wall“, „Horn“ und „HBF“. Anschließend suchen wir in diesem Trie nach „alt“. Für die exakte Suche erwarten wir „Altona“ und „Alter Wall“ als Antworten zurück, da beide Suchterme ein Präfix „alt“ enthalten. Die Grafik Abbildung 4.1 veranschaulicht den Ablauf.*



(a) Suchterm „ “

(b) Suchterm „a“

(c) Suchterm „al“

(d) Suchterm „alt“

Abbildung 4.1: Exakte Suche des Terms „alt“ in einem Trie

4.2.2 ICAN - Imcrementally Compute Active Nodes

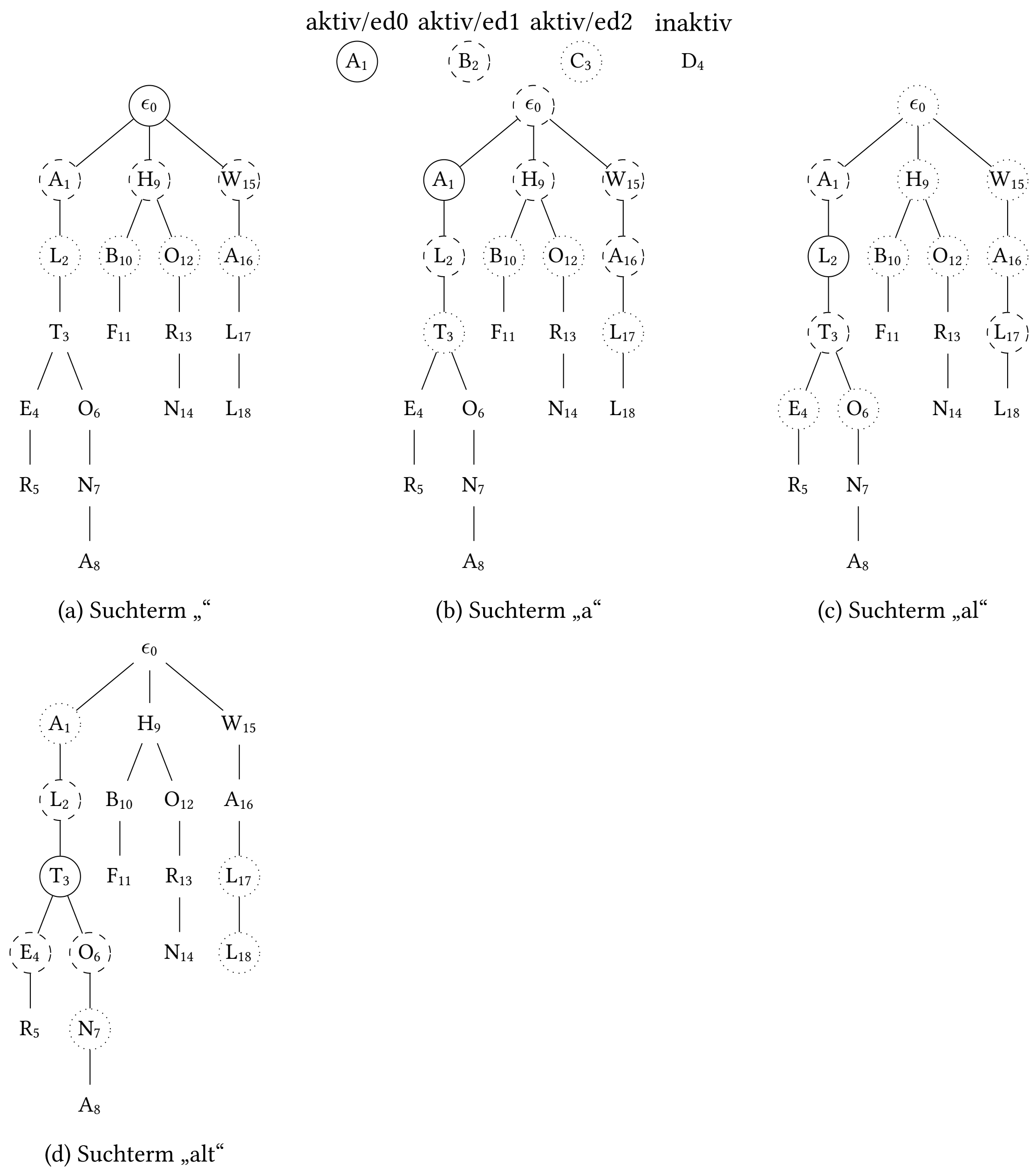
Bei der exakten Suche auf Basis des Trie-Index wie in Unterabschnitt 4.2.1 vorgestellt kann für eine Zeichenfolge s_x nur ein Knoten aktiv sein, da die Zeichenfolge einem eindeutigen Pfad in dem Trie entspricht. Werden nun jedoch Fehler innerhalb der Eingabe toleriert, kann es mehrere ähnliche Zeichenfolgen geben, die für eine Eingabe s_x relevant sein können, welche alle durch einen anderen Pfad im Trie repräsentiert werden. Entsprechend wären in einem Trie mehrere Knoten zugleich aktiv. Bevor wir die Berechnung aller relevanten Knoten näher beleuchten, wird nun der Begriff der *aktiven Knoten* eingeführt.

Defintion 2 (aktive Knoten) *Sei s eine Suchanfrage und τ die maximal berücksichtigte Editierdistanz, so wird ein Knoten n des Tries genau dann als* aktiver Knoten von s *bezeichnet, wenn die Editierdistanz $ed(s, n)$ zwischen Knoten n und der Anfrage s kleiner als τ ist, also $ed(s, n) \leq \tau$.*

Auch für den ICAN Algorithmus werden alle Schlüssel kanonisiert und in Schlüsselwörter zerlegt und wie schon für die exakte Suche in einem Trie gespeichert. Zu jeder Suchanfrage s gibt es eine Menge aktiver Knoten. Neben den aktiven Knoten selbst ist auch die dazugehörige Editierdistanz $ed(s, n)$ relevant. Deshalb wird zu jeder Anfrage s eine Menge von Tupeln der Form $\phi_s = \{(n, \xi_n)\}$ gespeichert, sodass n ein aktiver Knoten von s mit $\xi_n = ed(s, n) \leq \tau$ ist und jeder aktive Knoten von s in einem Tupel von ϕ_s vorkommt. Wird die Eingabe s nun fortgesetzt, wird die bisher gespeicherte Menge ϕ_s genutzt um die aktiven Knoten der neuen Eingabe zu berechnen.

Nach der Erstellung des Tries wird die Menge der aktiven Knoten ϕ_ϵ für die leere Eingabe ϵ berechnet. Dazu werden alle Knoten n, die zu einer Zeichenfolge mit einer Länge $|n| \leq \tau$ gehören zu ϕ_ϵ hinzugefügt, also: $\phi_\epsilon = \{(n, \xi_n) | \xi_n = |n| \leq \tau\}$. Wird nun eine Anfrage s gestellt, wird zuerst geschaut, ob es für ein Präfix von s bereits ein berechnetes Set im Cache gibt. Wenn nicht, muss das Ergebnis auf Basis von ϕ_ϵ inkrementell für alle Zeichen von s berechnet werden. Die Berechnung eines Sets $\phi_{s_{x+1}}$ für eine Eingabe $s_{x+1} = c_1 c_2 \ldots c_x c_{x+1}$ erfolgt nun auf Basis des vorhergehenden Sets ϕ_{s_x}:

Dafür wird in einem ersten Schritt das Set $\phi_{s_{x+1}}$ als leeres Set initialisiert. Anschließend wird für jedes Tupel $(n, \xi_n) \in \phi_{s_x}$ nun zuerst der Knoten n geprüft. Wir können das zu n gehörige Präfix p_x mit $\xi_n + 1$ Editieroperationen zu s_{x+1} transformieren, indem zuerst p_x mit ξ_n Editieroperationen zu s_x transformiert und dann der Buchstabe c_{x+1} an p_x angefügt wird. Sofern $\xi_n + 1 \leq \tau$ gilt, wird das Tupel $(n, \xi_n + 1)$ zu $\phi_{s_{x+1}}$ hinzugefügt. Anschließend folgt die Prüfung aller Kindknoten von n. Dabei gibt es zwei mögliche Fälle.



(a) Suchterm „“

(b) Suchterm „a“

(c) Suchterm „al“

(d) Suchterm „alt“

Abbildung 4.2: Suche des Terms „alt“ in einem Trie mit Hilfe des ICAN-Algorithmus bei maximaler Editier-Distanz $\tau = 2$

Der Buchstabe des betrachteten Kindknotens n_k von n ist c_{x+1}. In diesem Fall haben wir eine Übereinstimmung und wir können p_x mit ξ_n Editieroperationen in s_{x+1} überführen, indem zuerst mit ξ_n Editieroperationen p_x in s_x überführt wird und anschließend ohne weitere Editierkosten c_{x+1} mit dem Buchstaben von n_k gematcht wird. Folglich fügen wir das Tupel (n_k, ξ_n) zu $\phi_{s_{x+1}}$ hinzu. Zu beachten ist hier, dass noch Buchstaben am Ende der Zeichenfolge eingefügt werden können, sofern $\xi_n < \tau$ ist. Um auch die Einfügeoperationen zu berücksichtigen, wird für alle Nachfolger n_f von n_k, die maximal $\tau - \xi_n$ Buchstaben von n_k entfernt sind, zusätzlich das Tupel (n_f, ξ_{n_f}) mit $\xi_{n_f} = \xi_n + |n_f| - |n_k|)$ zu $\phi_{s_{x+1}}$ hinzugefügt, wobei $|n|$ die Länge der Zeichenfolge, die durch n beschrieben wird, bezeichnet.

Der andere Fall ist, dass der Buchstabe c_{n_k} von n_k ungleich c_{x+1} ist. In diesem Fall können wir s_{x+1} durch eine Substitution von c_{x+1} mit c_{n_k} für $\xi_n + 1$ Editieroperationen in p_{x+1} überführen. Dazu wird wieder zuerst p_x mit ξ_n Editieroperationen zu s_x transformiert und anschließend der Buchstabe c_{x+1} durch c_{n_k} ersetzt. Sofern $\xi_n + 1 \leq \tau$ gilt, wird das Tupel $(n_k, \xi_n + 1)$ zu $\phi_{s_{x+1}}$ hinzugefügt.

Nun kann es aufgrund verschiedener Folgen von Editieroperationen dazu kommen, dass für einen aktiven Knoten n bereits ein Tupel (n, ξ_1) in $\phi_{s_{x+1}}$ existiert, wenn wir ein Tupel (n, ξ_2) hinzufügen wollen. In diesem Fall behalten wir nur das Tupel mit der geringeren Editierdistanz und verwerfen das andere, da uns nur die minimale Editierdistanz interessiert.

Li u. a. (2011) zufolge weist die Berechnung des Sets aktiver Knoten $\phi_{s_{x+1}}$ auf Basis von ϕ_{s_x} eine Zeitkomplexität von $\mathcal{O}(\tau * (|\phi_s| + |\phi_{s-1}|))$ und eine Speicherkomplexität von $\mathcal{O}((|\phi_s| + |\phi_{s-1}|)$ auf, um auf Basis von s_x inkrementell die aktiven Knoten von s_{x+1} zu berechnen.

Wie auch bei der exakten Suche, können nun die Blätter aller aktiven Knoten traversiert werden, um die Vervollständigungen zu erhalten.

Beispiel 5 *Zur Visualisierung des Ganzen zeigt Abbildung 4.2 das Beispiel aus Unterabschnitt 4.2.1 unter Anwendung des ICAN-Algorithmus bei einer maximalen Editierdistanz von 2. Es ist schnell zu erkennen, dass eine hohe Zahl von Knoten zugleich aktiv ist und neben den erwarteten Pfaden für „Altona" und „Alter" durch das Erlauben einer Anzahl von Fehlern auch der Pfad für „Wall" aktive Knoten aufweist.*

Eines der Probleme dieses Verfahrens ist die hohe Anzahl an aktiven Knoten, da für jeden der Knoten überprüft werden muss, ob er und seine Kinder auch für die erweiterte Suchanfrage als Kandidaten in Frage kommen. Insbesondere bei kurzen Anfragen oder hohem Grenzwert τ ist meist eine erhebliche Anzahl von Knoten aktiv, da die Knoten in den ersten Ebenen oft sehr viele Nachfolger haben.

Der folgende Algorithmus nimmt sich dieses Problems an, indem er die Anzahl der aktiven Knoten reduziert:

4.2.3 ICPAN - Imcrementally Compute Pivotal Active Nodes

Der ICPAN Algorithmus ist eine Erweiterung des ICAN Algorithmus und funktioniert zunächst einmal grundsätzlich ähnlich. Auch hier wird mit Hilfe der Editierdistanz und eines Trie-Index imkrementell ein Set aktiver Knoten berechnet auf deren Basis sich dann die zu einer Anfrage passenden Vervollständigung aus den Kindern ableiten lassen. Der Unterschied liegt in den Details zur Berechnung der aktiven Knoten, was zu einer erheblichen Reduzierung dieser und damit einer deutlichen Verringerung der Suchzeit führt.

Der Unterschied des ICPAN Algorithmus gegenüber ICAN besteht darin, nur die Knoten zu speichern die *ausschlaggebend* für die daraus resultierende Liste an möglichen Vervollständigungen eines Wortes sind. So sind in Abbildung 4.2d die Knoten 4, 6 und 7 aktiv, obwohl diese eine höhere Editierdistanz aufweisen als einer ihrer Vorfahren und sie keinerlei zusätzliche Vervollständigungen zum Suchergebnis beitragen. Aus diesem Grund wird nun der Begriff der *ausschlaggebenden aktiven Knoten* eingeführt.

Defintion 3 (ausschlaggebende aktive Knoten) *Sei s eine Suchanfrage und τ die maximal berücksichtige Editierdistanz, so wird ein Knoten n des Tries genau dann als* ausschlaggebender aktiver Knoten von s *bezeichnet, wenn erstens n ein aktiver Knoten von s ist und zweitens es eine Transformation mit maximal τ Editieroperationen von s zum zu n gehörigen Präfix p gibt und die letzte Operation auf n (also dem letzten Buchstaben von p) eine Übereinstimmung ist.*

Es werden wieder alle Schlüsselwörter in einem Trie-Index gespeichert. Für jedes Schlüsselwort s_x einer Suchanfrage wird ein Set von Quadrupeln $\psi_{s_x} = \{(n, \xi_n^{s_x}, s_i, \xi_n^{s_i})\}$ gecached. n ist ein ausschlaggebender aktiver Knoten von s_x. s_i bezeichnet ein Präfix von s_x, dessen letzter Buchstabe mit dem Buchstaben von n übereinstimmt. Sollte es kein solches Präfix geben, ist $s_i = \epsilon$. Gibt es mehrere entsprechende Präfixe, entspricht s_i dem kürzesten dieser Präfixe. $\xi_n^{s_i}$ bezeichnet die Editierdistanz von dem Knoten n zu s_i und $\xi_n^{s_x}$ bezeichnet die Editierdistanz von n zu s_x. Sie setzt sich aus der Transformation von n zu s_i und der anschließenden Anfügung der Buchstaben nach p_i zusammen, sodass $\xi_n^{s_x} = \xi_n^{s_i} + |p_x| - |p_i|$ gilt.

Auch ICPAN beginnt mit dem Set für das leere Wort, welches in diesem Fall allerdings nur die Wurzel enthält: $\psi_\epsilon = \{(n_0, 0, \epsilon, 0)\}$. Wird einer beliebigen Anfrage $s_x = c_1 c_2 \dots c_x$ mit dem bereits berechneten Set ψ_{q_x} nun ein Buchstabe c_{x+1} angehängt, lässt sich das daraus resultierende Set $\psi_{q_{x+1}}$ nun wie folgt berechnen: Zuerst wird $\psi_{q_{x+1}}$ als leeres Set initialisiert. Anschließend wird für jedes Quadrupel $(n, \xi_n^{s_x}, s_i, \xi_n^{s_i})$ wieder zuerst der Knoten n untersucht und wir prüfen wie auch beim ICAN Algorithmus auf eine mögliche Löschoperation. Dafür wird s_{x+1} mit $\xi_n^{s_x} + 1$ Editieroperationen zu n transformiert, indem erst s_x mit $\xi_n^{s_x}$ Operationen

zu n transformiert und dann der letzten Buchstabe c_{x+1} gelöscht wird. Gilt $\xi_n^{s_x} + 1 \leq \tau$ fügen wir das Tupel $(n, \xi_n^{s_x} + 1, s_i, \xi_n^{s_i})$ zu $\psi_{s_{x+1}}$ hinzu.

Anschließend werden die Nachfolger von n betrachtet. Wir suchen unter den Nachfolgern von n alle Knoten, welche den Buchstaben c_{x+1} haben und innerhalb von $\tau - \xi_n^{s_i} + 1$ Schritten von n aus erreichbar ist. Die $+1$ ergibt sich daraus, dass die Operation auf dem letzten Buchstaben eine Übereinstimmung sein muss und somit die Editierdistanz nicht erhöht. Jeden dieser gefundenen Knoten n' können wir zu s_{x+1} transformieren, indem wir

1. n zu s_i transformieren,
2. die Buchstaben nach n und vor n' zu den Buchstaben $c_{i+1} \ldots c_x$ transfomieren und
3. den Buchstaben c_{x+1} mit dem Buchstaben von n' matchen.

Dafür werden $\xi_n^{s_{x+1}} = \xi_n^{s_i} + max(|n'| - |n| - 1, |s_x| - |s_i|)$ Operationen benötigt. Wenn $\xi_n^{s_{x+1}} \leq \tau$ ist, fügen wir $(n', \xi_n'^{s_{x+1}}, s_{x+1}, \xi_n'^{s_{x+1}})$ zu $\psi_{s_{x+1}}$ hinzu. Wir suchen also für jeden ausschlaggebenden aktiven Knoten in den Nachfolgern nach einem Knoten mit Übereinstimmung für den aktuellen Buchstaben von s_{x+1} innerhalb der verbleibenden Editierdistanz. Werden mehrere Quadrupel mit Bezug zum gleichen Knoten hinzugefügt, behalten wir wie auch bei ICAN nur den Knoten mit der geringeren Editierdistanz. Sollten beide Quadrupel für den Knoten die gleiche Editierdistanz haben, wird dasjenige Tupel behalten, dessen Editierdistanz für das Präfix $\xi_n^{s_i}$ geringer ist.

Im Anschluss müssen eventuell vorhandene nicht-ausschlaggebende aktive Knoten aus $\psi_{s_{x+1}}$ entfernt werden. Zu diesen kann es kommen, wenn es Transformationen zu s_{x+1} eines anderen ausschleggebenden aktiven Knotens gibt, die eine geringere Editierdistanz aufweisen, aber nicht mit einem Match enden. Um diese zu finden wird für jedes Quadrupel $(n, \xi_n^{s_x}, s_i, \xi_n^{s_i})$ von $\psi_{s_{x+1}}$ für das $s_i \neq s_{x+1}$ gilt nach Quadrupeln mit einem Vorfahren n_a von n mit $n_a \neq n$ gesucht. Werden solche Tupel $(n_a, \xi_{n_a}^{s_{x+1}}, s_a, \xi_{n_a}^{s_a})$ mit $\xi_{n_a}^{p_a} + max(|p_{x+1}| - |p_a|, |n| - |n_a|)$ gefunden, existiert somit eine Transformation von diesem Tupel aus, die günstiger ist, als von n aus. Somit wird $(n, \xi_n^{s_x}, s_i, \xi_n^{s_i})$ gelöscht.

Beispiel 6 *Abbildung 4.3 zeigt den Ablauf des Algorithmus anhand des gleichen Beispiels aus den letzten beiden Unterabschnitten. Durch die aktiven Knoten des IPCAN-Algorithmus werden die gleichen Pfade des Tries aktiviert und somit die gleichen Schlüsselwörter in das Ergebnis der Suche aufgenommen. Die Anzahl der Knoten ist allerdings deutlich geringer als bei dem ICAN Algorithmus. Der Unterschied ist vorallem bei dem leeren Suchwort ϵ deutlich zu sehen.*

Nach Li u. a. (2011) beträgt die Zeitkomplexität beim ICPAN Algorithmus für die inkrementelle Berechnung der ausschlaggebenden aktiven Knoten $\mathcal{O}(\tau * (|\psi_s| + |\psi_{s-1}|))$ und die

Speicherkomplexität $\mathcal{O}(|\phi_s| + |\phi_{s-1}|)$. Der ICPAN Algorithmus hat zwar die Anzahl der aktiven Knoten deutlich reduziert, doch kann abhängig von dem gewählten τ, der Größe des Alphabets und der Anzahl der Datensätze insbesondere bei kürzeren Suchanfragen immer noch eine erhebliche Anzahl von Knoten aktiv sein.

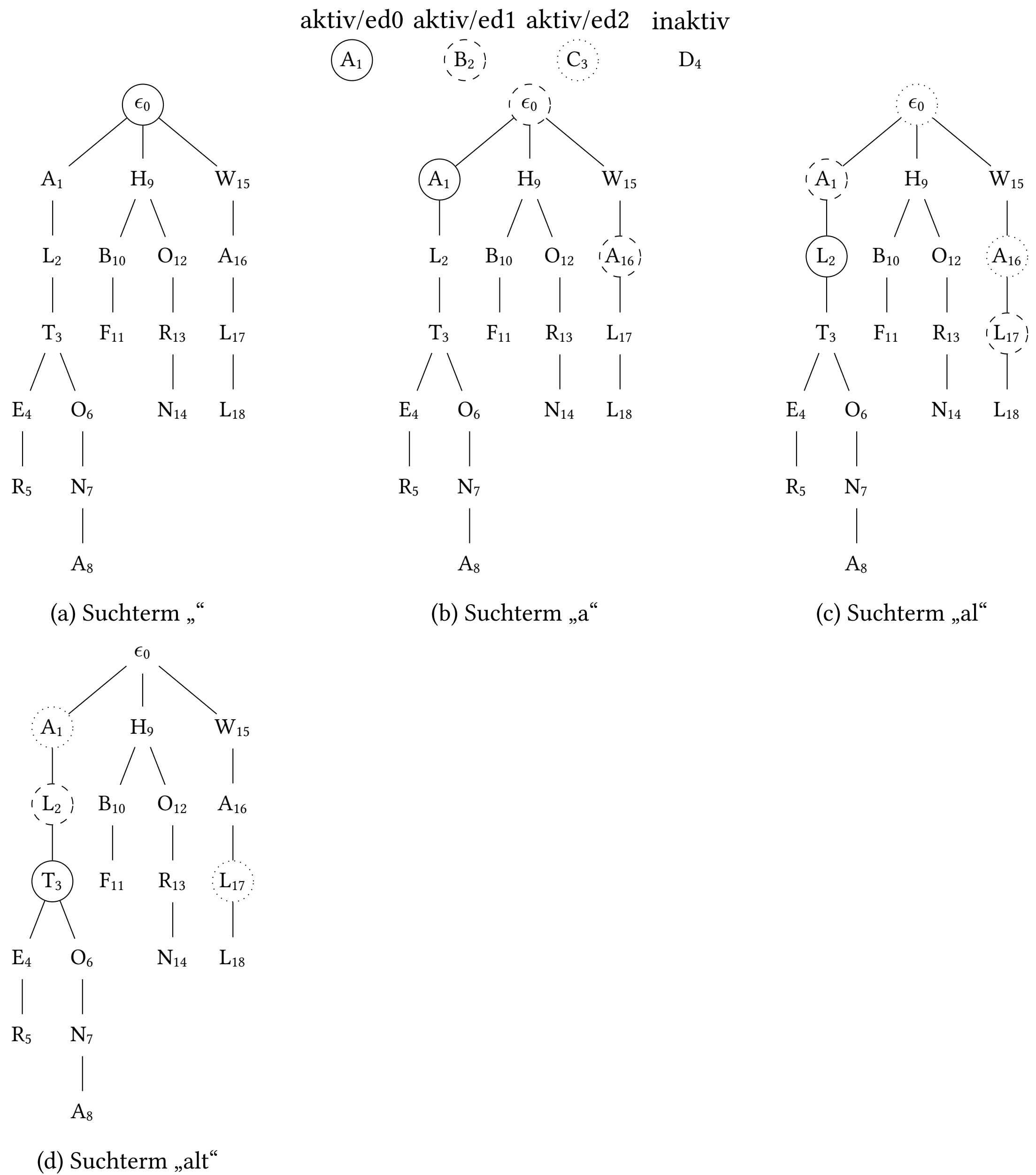


Abbildung 4.3: Suche des Terms „alt“ in einem Trie mit Hilfe des IPCAN-Algorithmus bei maximaler Editier-Distanz $\tau = 2$

4.2.4 IncNGTree - Incremental algorithm based on Neighborhood Generation and a Trie index

Der IncNGTree Algorithmus widmet sich der Problematik der noch immer hohen Anzahl an aktiven Knoten des ICPAN Algorithmus, indem die Trie-Index-Struktur mit der in Unterabschnitt 2.4.1 vorgestellten Löschungs-Nachbarschaft kombiniert wird. Ziel ist es, insbesondere kurze Suchanfragen stark zu beschleunigen, indem die Anzahl der aktiven Knoten auf Kosten eines größeren Index weiter reduziert wird.

Wie bereits ausgeführt, bezeichnet die Löschungs-Nachbarschaft einer Zeichenfolge s all jene Zeichenfolgen s', die durch Löschung von Zeichen aus der ursprünglichen Zeichenfolge generiert werden können. Diese Zeichenfolgen s' werden als *Variante von* s bezeichnet. Entsprechend bezeichnet eine k*-Variante von* s eine Zeichenfolge s', welche durch Löschung von k Zeichen aus s entstanden ist. //Die Überleitung in Mor-Fraenkel ist etwas... abrupt// Die Mor-Fraenkel Methode hat neben den Varianten von s auch die jeweiligen Listen der gelöschten Zeichen C^s und der Positionen der Löschungen D^s in einem Tripel (s', D^s, C^s) unter s' indiziert. Diese Listen werden genutzt, um falsch-positive Ergebnisse zu ermitteln, indem geprüft wird, ob die Löschliste D^s und sowie die Löschliste D^t $|D^s| + |D^t| - |D^s \cap D^t| \leq k$ erfüllen.

Beispiel 7 *Gegeben sind zwei Zeichenfolgen* $s = bahnhof$ *und* $t = bohnhf$ *mit einer Editierdistanz* $ED(s,t) = 2$*. Beide Zeichenfolgen haben eine gemeinsame Variante „bhnhf". Die Löschlisten sind entsprechend* $D^s = [2,5]$ *und* $D^t = [2]$.

$$\begin{aligned} k &= |D^s| + |D^p| - |D^s \cap D^p| \\ &= |[2,5]| + |[2]| - |[2]| \\ &= 2 + 1 - 1 \\ &= 2 \end{aligned}$$

Der IncNGTrie erzeugt, wie auch die Mor-Fraenkel Methode die Löschungs-Nachbarschaft aller Zeichenfolgen unseres Datenbestands, markiert jedoch alle Löschungen in den Varianten mit einem Sonderzeichen „#". Aus „Bhnhf", einer 2-Variante von „Bahnhof" wird somit „B#hnh#f". Diese markierten Varianten werden nun in einem Trie indiziert. Dies erlaubt uns, während der Traversierung zugleich inkrementell die Editierdistanz zu berechnen, da die Markierungen uns die benötigten Positionen von Löschungen aufzeigen.

Bevor das Suchverfahren vorgestellt wird, führen wir den Begriff des emphaktiven Zustands ein:

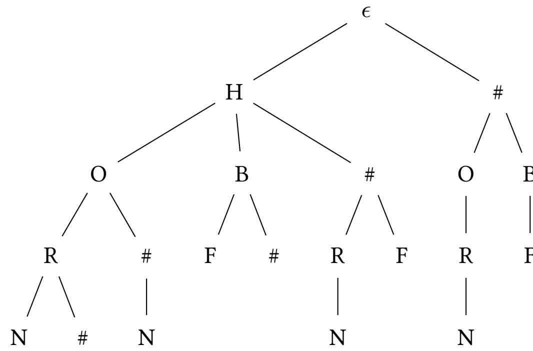


Abbildung 4.4: IncNGTrie der Worte „Horn“ und „HBF“ mit einer Editierdistanz von $\tau = 1$

Defintion 4 (Aktiver Zustand) *Ein aktiver Zustand ist ein Tripel* (n, u, δ)*.* n *ist ein Knoten des Tries.* u *ist ein curser, der anzeigt, dass das* u*-te Zeichen der Suchanfrage erwartet wird und* δ *repräsentiert die momentane Editierdistanz.*

Ein solcher aktiver Zustand repräsentiert eine Übereinstimmung von $q_[1 \ldots (u-1)]$ Zeichen mit einem Präfix einer Zeichenfolge des Tries, repräsentiert durch den Pfad von der Wurzel bis zum Knoten n mit einer Editierdistanz von δ. Wird die Suchanfrage q um den u-ten Buchstaben erweitert, werden aus den bisherigen aktiven Zuständen die neuen aktiven Zustände wie folgt berechnet:

Fall 1: Wenn n einen Kindknoten n' mit der Bezeichnung $q[u]$ hat, wird $(n', u+1, \delta)$ hinzugefügt.

Fall 2: Wir können Löschungen am Ende der Suchanfrage durchführen. Deshalb werden alle Kindknoten n' innerhalb der verbleibenden Editierdistanz $\tau - d_q$ hinzugefügt, also: $(n', u + d_q, \delta + d_q)$ mit $1 \leq d_q \leq \tau - d_q$.

Fall 3: Außerdem können Löschungen auf dem Datensatz durchgeführt werden. Somit können wir alle Knoten innerhalb der Editierdistanz hinzufügen, die über ein „#“ erreichbar sind. Daraus folgend wird $(n'', u + d_q, \delta + ma(d_q, d_s))$ ein aktiver Zustand, mit $1 \leq d_q, d_s, \leq \tau - \delta$.

Diese aktiven Zustände werden nun genutzt um eine Anfrage zu beantworten. Zu Beginn wird ein Set aktiver Zustände initialisiert. Dies enthält die Wurzel des Tries $(r, 1, 0)$, sowie alle Knoten, die von der Wurzel aus über maximal τ „#“ erreicht werden können. Wird nun die Eingabe q um $q[v]$ erweitert, werden die neuen aktiven Zustände aus den alten Zuständen berechnet. Alle Zustände mit $u > v$ bleiben aktiv, da das erwartete u-te Zeichen noch nicht eingegeben wurde. Für alle anderen aktiven Zustände werden alle Kindknoten mit $q[v]$ als Beschriftung aktiviert und anschließend alle Nachfolger eben dieser betrachtet, die über ein „#“ erreicht werden können. Wenn diese innerhalb, ähnlich der Initialisierung, innerhalb von $\tau - \delta$

Knoten erreichbar sind, werden sie ebenfalls aktiviert. Im letzten Schritt werden alle Blätter, die von den aktiven Knoten mit einem Cursor gleich $|q| + 1$ aus erreichbar sind betrachtet und die dazugehörigen Wörter als Ergebnis zurückgegeben.

Nach Xiao u. a. (2013) beträgt die Laufzeit $\mathcal{O}(\tau^2|q|^\tau)$ und die Speicherkomplexität $\mathcal{O}(\tau * (|q| + \tau)^\tau)$. Im Vergleich zu den beiden Verfahren ICAN und ICPAN wurde also die Abhängigkeit von der Größe des Alphabets aufgehoben. Dadurch wird die Laufzeit insbesondere für kurze Suchanfragen erhöht, da hier sehr viel weniger Zustände betrachtet und aktiviert werden müssen. Bei ICAN und ICPAN gibt es in den ersten Ebenen des Tries, abhängig vom gespeicherten Datensatz, oft für alle Zeichen des Alphabets Nachfolger, die betrachtet und aktiviert werden, während für den IncNGTrie lediglich der übereinstimmende Pfad, sowie Pfade mit „#“ als Übergängen betrachtet werden. Die Schwäche dieses Verfahrens liegt hier in dem verhältnismäßig größeren Index, der bei einer Editierdistanz von 2 bereits je nach Datensatz bereits eine 6 bis 8-fachen Speicherbedarf aufweist.

4.2.5 META - Matching-based framework for Error-Tolerant Autocompletion

META ist ebenfalls ein Trie-basierter Algorithmus, der vom grundlegenden Vorgehen dem IPCAN Algorithmus ähnelt. Der ICPAN Algorithmus betrachtet nur ausschlaggebende aktive Knoten, also Knoten, die eine Übereinstimmung zweier Buchstaben q_i und s_j zwischen der Suchanfrage und einem Datensatz haben. Auch META betrachtet diese Knoten und nutzt sie in einem *Matching Set* um die Editierdistanzen zwischen einer Eingabe und einem Präfix zu berechnen. Zuerst wird der Begriff des Matchings eingeführt:

Defintion 5 (Matching) *Gegeben seien zwei Zeichenfolgen q und s. Ein* Matching *bezeichnet ein Tripel $m = (i, j, k)$ mit einer Übereinstimmung von zwei Buchstaben $q_i = s_j$ mit $1 \leq i \leq |q|$ bzw. $1 \leq j \leq |s|$ bei einer Editier-Distanz $k = ed(q_i, s_j)$.*

Von einem Matching kann die Editier-Distanz abgeleitet werden, indem die im Matching vermerkte Editier-Distanz $m.k$ mit der größeren, verbleibenden Länge der beiden zu vergleichenden Wörter addiert wird, also: $ed_m(q, s) = m.k + \max(|q| - m.i, |s| - m.j)$.

Ein Matching Set $M(q_x, s)$ enthält nun alle Matchings zwischen q_x und s. //wie sieht diese Erkenntnis in META aus? zumindst etwas näher ausführen wäre gut, nur leider fehlt in dem Paper der Beweis// Dank Deng u. a. (2016) wissen wir, dass die Editier-Distanz zweier Zeichenfolgen q_i und s_j der minimalen abgeleiteten Editier-Distanz der Matchings aus $M(q_{i-1}, s)$ entspricht. Für alle Matchings zweier Zeichenfolgen q und s gilt dementsprechend

$ed(q_i, s_j) = m \in M(q_i, s)|t \min(m.k + \max(|q_i| - m.i, |s_j| - m.j))$. Auf Basis des Matching Sets lässt sich somit die Editierdistanz zwischen zwei Worten inkrementell berechnen.

Zuerst wird ein Matching Set $M(q_0, s)$ mit $m_{q_0,s_0} = (0, 0, 0)$ initialisiert. Anschließend betrachten wir für jeden Buchstaben $q_i, 1 \leq i \leq |q|$ alle Matchings $m = (i', j', k')$ des Matching Sets $M(qi - 1, s)$ und suchen in s nach Übereinstimmungen von q_i mit $s_j, j > j'$. Für all diese Übereinstimmungen wird die Editier-Distanz berechnet und in einer Map H mit j als Schlüssel gespeichert. Finden wir für ein j mehrere Editier-Distanzen, speichern wir die Geringste und verwerfen die anderen. Anschließend können wir aus der Map H Matchings der Form $m = (i, j, H[j])$ erzeugen. Das Matching Set $M(q_i, s)$ setzt sich nun aus allen Matchings aus $M(q_{i-1}, s)$, sowie allen zuvor erzeugten Matchings zusammen.

Der Matching Set Ansatz lässt sich auch auf Tries übertragen. Im Trie entspricht s_j einem Knoten des Tries n_j, wobei j die Tiefe des Knotens im Trie darstellt. Daraus lassen sich dann die Begriffe des *aktiven Matchings und aktiven Knotens* ableiten:

Defintion 6 (aktives Matching und Aktiver Knoten) *Gegeben sei eine Zeichenfolge q und eine maximale Editier-Distanz τ. $m = (i, n, k)$ ist ein aktives Matching von q und n ist ein aktiver Knoten von q, sofern $ed_m(q, n) \leq \tau$ innerhalb der maximalen Editier-Distanz liegt.*

Für einen Trie $\mathcal{T}$ läuft die Berechnung des Matching Sets analog zum oben vorgestellten Algorithmus ab. Zuerst wird ein Matching Set $M(q_0, \mathcal{T})$ mit $m = (0, n_0, 0)$ initialisiert. Anschließend betrachten wir für jeden Buchstaben $q_i, 1 \leq i \leq |q|$ alle Matchings $m = (i', n', k')$ des Matching Sets $M(qi - 1, \mathcal{T})$ und suchen in den Nachfolgern von n' nach Knoten n mit q_i als Beschriftung innerhalb der Editier-Distanz τ. Für alle gefundenen Knoten n wird die Editier-Distanz $ed_m(q_i, n) = m.k + \max(i - m.i, |n| - |n'|)$ berechnet und in einer Map H mit n als Schlüssel gespeichert. Auch hier speichern wir nur die niedrigst gefundene Editier-Distanz für jeden Knoten. Anschließend können wir aus der Map H Matchings der Form $m = (i, n, H[n])$ erzeugen. Das Matching Set $M(q_i, \mathcal{T})$ setzt sich nun aus allen Matchings aus $M(q_{i-1}, \mathcal{T})$, die innerhalb der Editier-Distanz zu q_i sind, sowie allen zuvor erzeugten Matchings zusammen.

Am Ende können aus dem Matching Set alle resultierenden Autovervollständigungen extrahiert werden, indem ähnlich der anderen Verfahren alle Blatt-Nachfolger der Knoten, die in den Matchings enthalten sind, iteriert und die Wörter die sie repräsentieren gesammelt werden.

Um die Suche nach Nachfolger-Knoten mit einem bestimmten Buchstaben zu beschleunigen, werden alle Knoten des Tries in pre-order mit einer id, beginnend mit 1, versehen. Außerdem wird an jedem Knoten ein Interval aller Ids seiner Nachfolger gespeichert. Darüber hinaus

wird ein zwei-dimensionaler invertierter Index $\mathcal{I}$ aufgebaut, wobei jede Liste $\mathcal{I}[tiefe][x]$ alle Knoten n mit $|n| = tiefe$ und x als Beschriftung enthält. Diese Listen sind nach den Ids der Knoten sortiert, um die binäre Suche nach Knoten zu ermöglichen.

Bei genauerer Betrachtung lässt sich feststellen, dass die bisherige Vorgehensweise der des IPCAN-Algorithmus. Die Unterschiede liegen zum Einen in der Verwendung des Matching-Sets zur Bestimmung der Editierdistanz und zum Anderen in der Verwendung des Inverted-Node-Index zur effizienteren Suche nach Knoten.

4.3 Bilden der Schnittmenge

Für eine Suchanfrage Q mit den Suchtermen $p_1...p_x$ stellt $k_{i_1}, ...k_{i_y}$ die Liste der zu einem Präfix p_i gehörenden Schlüsselwörter dar. L_{i_j} bezeichnet die Invertierte Liste von k_{i_j}, also die Liste aller dazugehörigen Datensätze und $U_i = \cup_j L_{i_j}$ die Vereinigung aller invertierten Listen eines Suchterms. Für die Berechnung die Schnittmenge aller Vereinigungen $U_1, ..., U_x$ müssen zuerst die invertierten Listen gefunden und aus Ihnen die Vereinigung gebildet werden. Anschließend muss die Schnittmenge dieser Vereinigungen berechnet werden, um so nur die Datensätze zu erhalten, die zu allen Suchtermen korrelieren.

Um die invertierten Listen zu erhalten, können diese entweder direkt im Trie hinterlegt werden oder in einer Map, die zu jedem Schlüsselwort die Datensätze speichert. Die Zeitkomplexität zur Berechnung der Vereinigung U_i beträgt dann etwa $\mathcal{O}(\sum_{i,j} |L_{i_j}|)$, wobei bei kürzeren Suchtermen die Anzahl der korrelierenden Schlüsselwörter und damit der invertierten Listen potenziell ansteigt, da zu kürzere Präfixe potenziell in einer größeren Menge an Schlüsselwörtern vorkommen. Der benötigte Speicherplatz für die invertierten Listen beträgt etwa $\mathcal{O}(n * L)$, wobei n die Anzahl der Datensätze und L die durchschnittliche Anzahl von Schlüsselwörtern der Datensätze ist. Um die Berechnung der Vereinigungen zu beschleunigen können diese auch vorberechnet und für jedes Präfix gespeichert werden, allerdings erhöht dies den Speicherbedarf erheblich. Hier beträgt die Speicherkomplexität $\mathcal{O}(n * L * w)$. w entspricht der durchschnittlichen Länge der Schlüsselwörter (**?**).

Anschließend muss die Schnittmenge aller Vereinigungen gebildet werden. Die obere Laufzeitgrenze beträgt dafür $\mathcal{O}(\prod_x |U_x|)$. Oft ist die Laufzeit jedoch geringer, da, wenn ein Element in einer Liste nicht auftaucht, es bereits kein Element der Schnittmenge mehr sein kann und die anderen Listen nicht mehr nach diesem Element durchsucht werden müssen.

? bildet die Schnittmenge $\cap U_i$ mit Hilfe von sogenannten „Forward Lists“. Außerdem wird durch das Speichern zusätzlicher Informationen im Trie die Berechnung einiger Vereinigungen umgangen, da die Vereinigungen in dem vorgestellten Verfahren nicht explizit materialisiert

werden müssen. Dazu werden alle Schlüsselwörter mit einer ID versehen, wobei die Vergabe der IDs entsprechend der alphabetischen Ordnung entspricht. Anschließend wird in jedem Knoten ein Interval $[minID, maxID]$ hinterlegt, welches alle IDs von Schlüsselwörtern enthält, welche durch dieses Präfix repräsentiert werden. Zuletzt muss die Länge der Vereiningung eines Knotens (bzw. dessen Präfix) einmal vorberechnet und in den Knoten gespeichert oder auf Basis der Anzahl von Schlüsselworten eines Knotens geschätzt werden. Desweiteren benötigen wir die bereits erwähnten „Forward Lists". Bei den Forward Lists handelt es sich um eine Map, welche jedem Datensatz die IDs der dazugehörigen Schlüsselwörter zuordnet. Die Liste der IDs ist dabei sortiert. Auf Basis dieser Informationen können die Vereinigungslisten nach der Anzahl der in ihr enthaltenen Elemente sortiert werden, ohne diese zu materialiseren. Um nun die Datensätze zu traversieren, werden die aktiven Knoten des Baumes für ein Präfix durchlaufen und aus den Nachfolgern die hinterlegten Datensätze durchlaufen. Um nun einen Datensatz r zu daraufhin zu überprüfen, ob er in der (nicht materialisierten) Vereinigung U_k eines Präfix p_k enthalten ist, können wir stattdessen r daraufhin untersuchen, ob das Präfix p_k in einem der Wörter aus der Forward List von r enthalten ist. Dafür iterieren wir jeden aktiven Knoten von p_x und untersuchen die Forward List daraufhin, ob r eine Wort-Id enthält die innerhalb des ID-Intervals eines der aktiven Knoten ist von p_x ist. Für die Überprüfung eines einzelnen aktiven Knotens führen wir dabei zuerst eine binäre Suche nach der unteren Grenze $minID$ des Intervals aus und überprüfen ob das nächste Element in der Forward List nicht größer $maxID$ des Intervals ist. Diese Suche weist pro Datensatz eine Zeitkomplexität von $\mathcal{O}((l-1)\log L)$ auf, wobei l die Anzahl der Schlüsselwörter einer Anfrage und L die durchschnittliche Anzahl von Schlüsselwörtern pro Datensatz bezeichnet. Auf diese Weise können wir nun jeden Datensatz auf die Zugehörigkeit zur Schnittmenge überprüfen.

4.4 Priorisierung der Kandidaten

Nachdem wir nun eine Liste von Kandidaten $r_1, r_2, ..., r_x$ haben, müssen wir diese nach einer definitierten Priorität sortieren. Bei der Priorisierung stellen sich mehrere Fragen.

1. Welche Informationen stehen zur Verfügung die wir in die Priorisierung einfließen lassen können?
2. Welche Informationen haben Einfluss auf die Priorisierung?
3. Wie stark fließen die verschiedenen Einflüsse in die Priorisierung ein?
4. Sind die Einflussfaktoren untereinander strikt priorisiert oder werden sie additiv oder multiplikativ zu einem Gesamtgewicht berechnet?

Für die Priorisierung haben wir einige verschiedene Daten zur Verfügung. Zum einen sind da Informationen über die jeweiligen Lokationen, unsere Datensätze, selbst. Jede Lokation beinhaltet Daten darüber, in welchem Ort die Lokation liegt, als auch was für eine Art von Lokation es ist. Aus anderen Teilen des Geofox Projektes könnten nun auch weitere Daten über die Lokationen eingeholt werden. Im Falle von Stationen könnten zum Beispiel Bahnhöfe gegenüber Busstationen oder der U-Bahn höher priorisiert werden oder man könnte die Anzahl der Verbindungen, die über diese Station fahren mit einfließen lassen. Desweiteren liegen uns Log Files über bisherige Suchen vor, welche unter Anderem Aufschluss darüber geben, wie oft bestimmte Lokationen als Ziel ausgewählt wurden, was uns widerum das Ableiten der Beliebheit oder auch Beliebtheitstrends verschiedener Lokationen ermöglicht. Außerdem haben wir Informationen über die Länge der Suchanfrage und die Länge der Lokationsbezeichnung, sowie natürlich die Editierdistanz über die eine Lokationen gefunden wurde.

Einige diese Daten sind zwingend notwendig um bestimmte Anforderungen einhalten zu können. So benötigen wir die Art der Lokation für die Anforderungen A5.1, A5.2 und A4.2, die Suchanfrage selbst für A5.2 und den Ort der Lokation für A4.1. Zugleich ist die Editierdistanz insofern relevant, als dass die Lokation, dessen Bezeichnung genauer auf unsere Suchanfrage zutrifft, auch wahrscheinlicher das gewünschte Ziel des Nutzers ist.

5 Konzeption und Implementierung der Suchmaschine

Im Folgenden werden einige der Vor- und Nachteile der vorgestellten Algorithmen nocheinmal betrachtet. Darauf aufbauen wird ein Konzept für die Suchmaschine innerhalb der Geofox-Anwendung vorgestellt.

Die im vorherigen Kapitel vorgestellten Algorithmen bauen teilweise aufeinander auf. Der ICAN-Algorithmus hat insbesondere bei kurzen Anfragen eine sehr hohe Zahl aktiver Knoten mit vielen redundanten und, wie die durch den IPCAN eingeführten Verbesserungen zeigen, zum Teil irrelevanten Berechnungen. Der IPCAN Algorithmus reduziert die Anzahl der aktiven Knoten im Vergleich zu ICAN erheblich und beschleunigt dadurch die Berechnung der Kanididaten insbesondere für kürzere Suchanfragen. Der IncNGTrie reduziert zwar die Berechnungszeit weiter, indem hier das Problem der unscharfen Suche auf mehrere exakten Suchen heruntergebrochen wird, hat allerdings den Nachteil des deutlich größeren Index. Der erhöhte Speicherbedarf ist insbesondere im Kontext von Geofox relevant, da die Autovervollständigung nur einen kleinen Teil des Systems ausmacht und der Speicher auch für andere Komponenten, wie beispielsweise der Routenfindung, benötigt werden. Der META-Algorithmus funktioniert in seiner Grundform ähnlich des IPCAN-Algorithmus, enthält allerdings einige Optimierungen, die die Berechnung von Nachfolgern insbesondere für höhere Editierdistanzen beschleunigt, sodass META mit einem nur minimal größeren Index als IPCAN eine geringere Laufzeit für Suchanfragen aufweist.

Auf Basis dessen habe ich mich entschieden, den META Algorithmus als Basis für die Suchmaschine zu verwenden.

Im Folgenden wird der Aufbau der Suchmaschine beschrieben. Dabei gehe ich zuerst auf die verwendeten Technologien und die Einbettung in das bestehende System ein. Darauf folgend wird die interne Struktur, sowie der interne Ablauf innerhalb der Komponente beschrieben.

Die Suchmaschine wurde vollständig in Java implementiert und verwendet keine externen Frameworks. Während der Initialisierung holt sich die Komponente die zu indizierenden Lokationen aus verschiedenen anderen Strukturen innerhalb des Geofox-Projekts, namentlich

dem VillageManager für Addressen, dem POIFinder für POIs und einem DataManager sowie dem StationFinder für Stationen und deren Aliasnamen.

Das Interface OneFieldSuchmaschine dient als Schnittstelle, damit die alte n-Gramm basierte Suchmaschine ohne Anpassungen an anderen Teilen des Geofox Projekts durch die META-Suchmaschine ersetzt werden kann. Entsprechend übernmmt die METAOneFieldSearchEngine zum Einen das Füllen des Index mit den benötigten Daten sowie die Delegation der Suchanfragen an den eigentlichen Index. Die Klasse METASearchEngine stellt die eigentliche Suchmaschine als solche dar. Sie dient als Fassade der Komponente nach außen und stellt die grundlegenden Funktionalitäten zur Initialisierung, Suche und Zustandsabfrage bereit. Intern besteht die Suchmaschine aus mehreren Elementen. Zur Speicherung der Schlüsselwörter benutzt sie einen Trie, sowie den in Unterabschnitt 4.2.5 vorgestellten Inverted Node Index, welcher die Suche nach Nachfolgeknoten innerhalb des Tries beschleunigt. Intern besteht der Inverted Node Index aus einer Liste von Maps, welche jedem Buchstaben widerum eine Liste von Knoten des Tries zuordnet. Der Index der ersten Liste entspricht dabei immer der Tiefe des Knotens und der Buchstabe der Beschriftung des Knotens. Die Datensätze selbst werden in einer Multimap gespeichert, welche zu einem Schlüsselwort eine Liste aller korrelierenden Datensätze speichert. Ein auf regulären Ausdrücken basierender Kanonisierer standardisiert sämtliche Suchterme und Schlüsselwörter. Der letzte interne Baustein ist dann der RecordSorter, welcher für die Filterung und Sortierung der gefundenen Lokationen zuständig ist. Die Suchmaschine ist dabei generisch aufgebaut, sodass durch die Bereitstellung eines entsprechenden Sortierers beliebige Objekte indiziert werden könnten. Bei Initialisierung der Suchmaschine wird außerdem ein Divisor mitgegeben, auf Basis dessen sich dann die Editierdistanz für die Suche nach Schlüssselwörtern im Trie errechnen lässt. Abbildung 5.1 visualisiert noch einmal den Aufbau der Suchmaschine.

Die Initialisierung des Indizes funktioniert wie folgt: Bei Erzeugung einer neuen METASearchEngine Instanz wird intern ein leerer Trie, ein dazugehöriger, leerer Inverted Node Index, sowie eine leere Multimap für die Zuordnung von Schlüsselwörtern zu Datensätzen erzeugt. Darauf folgend werden sämtliche zu indizierende Lokationen zusammen mit einem Schlüssel an die Suchmaschine übergeben. Der Schlüssel wird zuerst kanonisiert und anschließend in einzelne Schlüsselwörter zerlegt. Jedes dieser Schlüsselwörter wird nun im Trie eingefügt. Außerdem werden die Lokationen unter allen dazugehörigen Schlüsselwörtern in die Multimap eingefügt. Sobald alle Lokationen indiziert wurden, wird der Index finalisiert. Während der Finalisierung wird der Trie in pre-order und in post-order traversiert. In pre-order wird jeder Knoten inkrementell mit einer eindeutigen, numerischen ID versehen und anschließend im Inverted Node Index eingefügt. Während der post-order-Traversion wird in den Knoten das

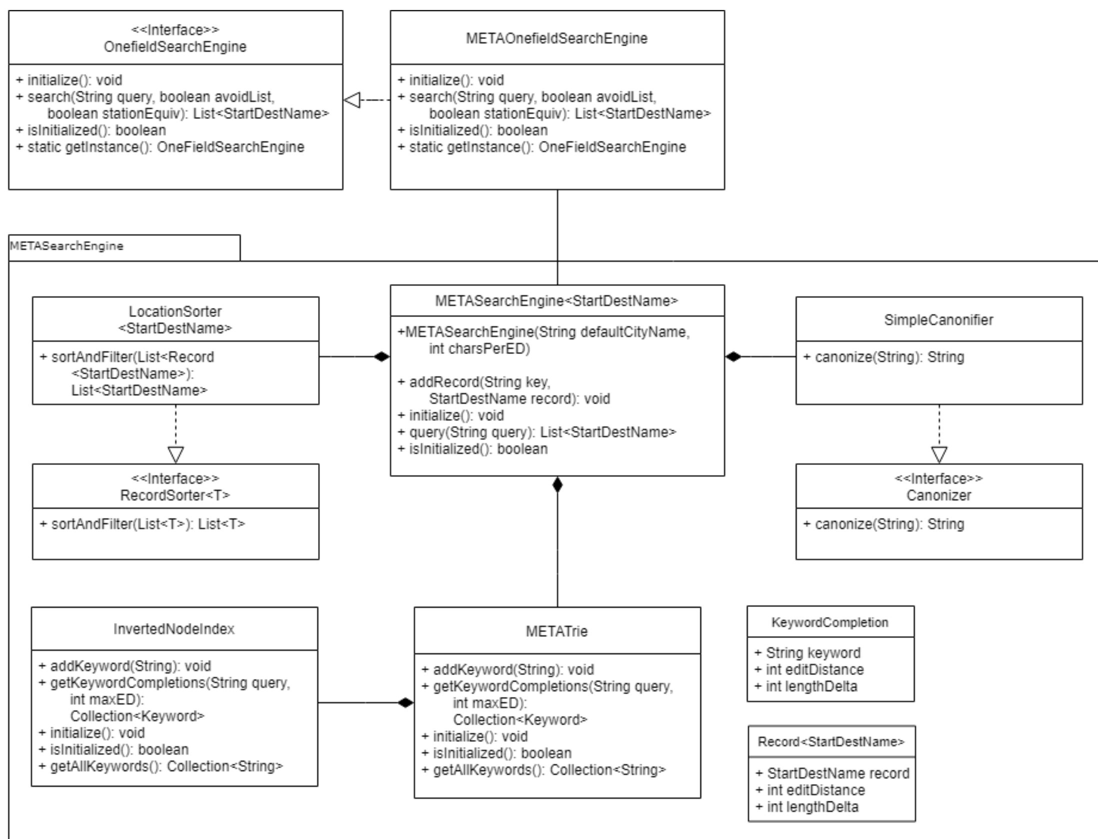


Abbildung 5.1: UML-Klassendiagramm: METASuchmaschine

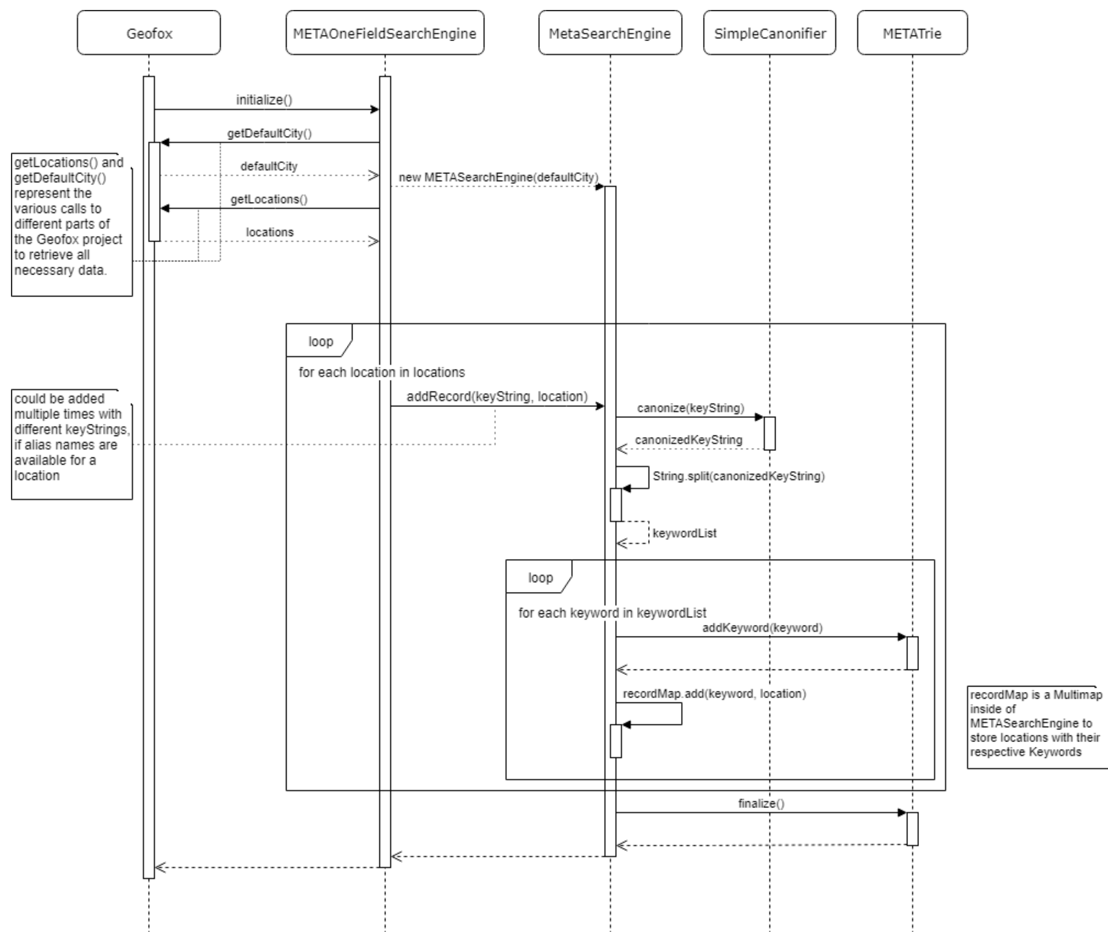


Abbildung 5.2: UML-Sequenzdiagramm: Initialisierungsphase der METASuchmaschine

Interval mit den IDs aller im Unterbaum befindlichen Knoten gespeichert. Die Suchmaschine ist nun bereit, Suchanfragen zu beantworten. Die Indizierung wird durch Abbildung 5.2 noch einmal veranschaulicht.

Werden nun Suchanfragen gestellt, werden diese wie in Abbildung 5.3 dargestellt wie folgt beantwortet: Zuerst wird die Suchanfrage kanonisiert und in die einzelnen Suchterme zerlegt. Anschließend wird der Trie nach dem Schlüsselwort durchsucht. Dafür wird zuerst die Editierdistanz ermittelt, indem die Länge des Schlüssels durch den während der Initialisierung übergebenen Divisors dividiert wird. Das Ergebnis stellt die Editierdistanz für die anschließende Suche im Trie dar. Eine detaillierte Beschreibung des Ablaufs dieser Suche befindet sich bereits in Unterabschnitt 4.2.5. Nachdem für jeden Suchterm eine Liste von Schlüsselwörtern zurückgegeben wurde, werden diese nun genutzt um die zu den Suchtermen passenden Loka-

tionen aus der Multimap zu erhalten Dabei wird für jedes ursprüngliche Schlüsselwort der Suchanfrage eine Liste mit allen zu dessen Vervollständigungen korrelierenden Lokationen erstellt. Aus den entstandenen Listen von Lokationen wird nun die Schnittmenge gebildet, sodass wir nur noch Lokationen behalten, die mit allen der Suchterme korrelieren. Dafür wird die Schnittmenge der kürzesten mit allen anderen Listen gebildet. Zuletzt müssen die gefundenen Lokationen noch sortiert werden.

Die Sortierung erfolgt in einem Stufensystem, d.h. dass alle Elemente erst nach einem Kriterium sortiert werden und nur bei gleicher Priorisierung das nächste Kriterium in Betracht gezogen wird. Zuerst werden alle Lokationen nach der Editierdistanz unter der sie gefunden wurden sortiert. Haben mehrere Lokationen die gleiche Editierdistanz zum Suchbegriff, werden Orte innerhalb Hamburgs bevorzugt. Bahnhöfe werden gegenüber anderen Lokationen bevorzugt. Sollte die Suchanfrage Ziffern enthalten, werden Straßen gegenüber Stationen und POIs bevorzugt, andernfalls werden die Straßen gegenüber den anderen beiden Typen benachteiligt. Ist die Sortierung abgeschlossen, wird die sortierte Liste als Ergebnis an den Nutzer zurückgegeben.

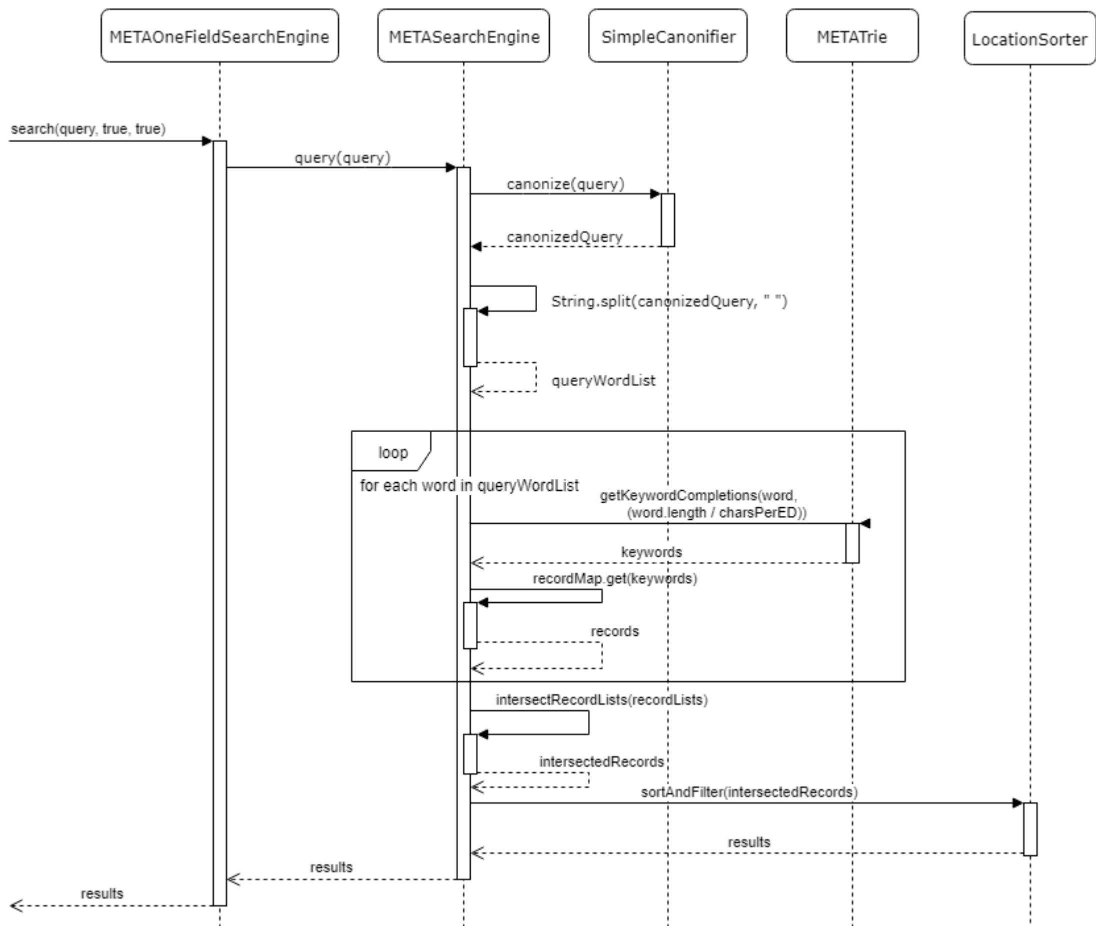


Abbildung 5.3: UML-Sequenzdiagramm: Bearbeitung einer Suchanfrage an die METASearchEngine

6 Tests und Evaluation

In diesem Kapitel wird die Implementierung des Algorithmus und des Priorisierungs-Verfahrens näher beschrieben. Außerdem werden verschiedene Testverfahren vorgestellt, um im Anschluss die Effizienz und die Skalierbarkeit des Systems zu untersuchen.

6.1 Tests

Wir werden im Folgenden das Laufzeitverhalten der implementierten Suchmaschine analysieren. Dabei betrachten wir insbesondere die Laufzeiten der folgenden Teilaufgaben: Indizierung, Suche nach Datensätzen und Priorisierung der Ergebnisse. Darüber hinaus werden wir auch den Einfluss der Länge der Suchanfrage, der Größe des Indexes und der Anzahl der Wörter auf die Laufzeit, sowie den Recall untersuchen. Diese Messungen werden auf zwei Datensätzen durchgeführt:

HVV-Lokationsdaten Dieser Datensatz umfasst insgesamt 61.861 Lokationen (15.683 Stationen, 41.100 Straßen, 5.078 POIs), welche unter insgesamt 103.156 Namen indiziert werden (da für einige Stationen und POIs mehrere Bezeichnungen existieren).

DBLP Der DBLP Datensatz umfasst insgesamt 1.865.774 wissenschaftliche Artikel bestehend aus insgesamt 1.840.188 unterschiedlichen Worten und Autorennamen (es wurden jeweils nur die Titel und Autoren der Artikel als Schlüsselwörter verwendet).

Während die HVV-Daten am Ende für die eigentliche Größe des Index und die Laufzeit während des Betriebs von Geofox entscheidend sind, wird der erheblich größere Datensatz des DBLP lediglich dafür genutzt, auch die Skalierbarkeit der Suchnmaschine bei größeren Datenmengen abschätzen zu können, da mir landesweite Straßen und Stationsdaten leider nicht zur Verfügung stehen.

Alle Tests werden auf einem Laptop mit einer Intel Core i7-7500U, 2,9 - 2.9GHz CPU und 16GB Arbeitsspeicher und Windows 10 Pro 64 bit durchgeführt.

Als erstes erfolgt die Betrachtung der Laufzeit für die Erstellung des Index. Dafür werden 1.000, 5.000, 10.000, 50.000, 100.000, 500.000 und 1.000.000 Datensätze der DBLP Daten indiziert und die benötigte Laufzeit gemessen.

Anschließend erfolgt die Analyse der Laufzeit des Algorithmus. Dafür werden Suchanfrage unterschiedlicher Länge und verschiedener Anzahl von Suchtermen auf Indizes unterschiedlicher Größe gestellt, um den Einfluss dieser Faktoren auf die Laufzeit untersuchen zu können. Außerdem untersuchen wir den Einfluss verschiedener Editierdistanzen, indem wir den bei Initialisierung übergebenen Divisor beeinflussen. Für jede Parameterkonstellation werden 1000 Suchanfragen gestellt und die durchschnittliche Laufzeit berechnet. Die Suchanfragen werden dabei aus den Datensätzen erzeugt, indem 1000 Datensätze zufällig ausgewählt und entsprechend der Parameter vorverarbeitet werden.

Außerdem betrachten wir die Größe des Recalls.

Für den Vergleich des NGram-basierten Verfahrens mit der META-Suchmaschine führen wir außerdem Messungen mit unterschiedlichen Suchanfragen auf dem vollständige HVV-Datensatz durch.

6.2 Evaluation

6.2.1 Analyse der META-Suchmaschine

Im Folgenden wird die META-Suchmaschine auf die Laufzeit zur Indizierung von Datensätzen unterschiedlicher Größe betrachtet. Außerdem wird das Laufzeitverhalten bei Suchanfrgaen unterschiedlicher Länge von 4 bis 10 Zeichen sowie bei Suchanfragen mit einem bis vier Suchtermen untersucht. Abschließend betrachten wir die Laufzeit und die Größe des Recalls bei variierender Länge der Suchanfrage und verschiedenen Editierdistanzwerten.

Abbildung 6.1 zeigt die durchschnittliche Laufzeit für die Indizierung von 10 bis 1000000 Elementen des DBLP Datensatzes nach 1000 Durchläufen. Die Y-Achse zeigt die Laufzeit in Millisekunden und die X-Achse die Anzahl der zu indizierenden Elemente. Für 10 bis 1000 Elemente unterliegt der zeitliche Verlauf einigen Schwankungen, ab 1000 Elementen bis 1000000 Elementen allerdings ist mit 230ms und 23240ms respektiveein linearer Anstieg der Laufzeit zur Anzahl der Elemente zu erkennen.

In Abbildung 6.2 wird die Laufzeit der Größe des Index bei Suchanfragen der Länge 6 betrachtett. Auch hier ist auf der Y-Achse die Laufzeit in Millisekunden, sowohl die Anzahl der indizierten Elemente auf der X-Achse abgebildet. Es sind drei Graphen dargestellt: Der hellbalue Graph zeigt die Zeit für die Suche von Schlüsselwörtern und den dazugehörigen Datensätzen innerhalb der Suchmaschine. Der dunkelblaue Graph repräsentiert die benötigten

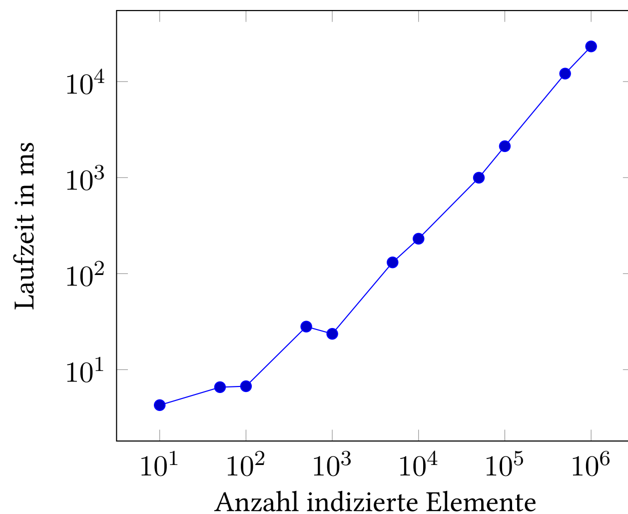


Abbildung 6.1: Indexerstellung - DLBP Laufzeit
Laufzeit zur Erstellung des Index für META auf Basis des DBLP-Datensatzes.

Zeitanteil für die anschließende Sortierung der Elemente. Der dritte, blaue Graph repräsentiert die Gesamtzeit der Suche. Es zeigt sich, dass die Sortierung der Elemente hier nur etwa ein viertel bis sechstel der Gesamtzeit ausmacht. Auch hier ist ein linearer Verlauf der Laufzeit in Abhängigkeit von der Indexgröße zu erkennen.

Abbildung 6.3 und Abbildung 6.4 zeigen den Einfluss der Wortlänge und der Anzahl der Worte auf die Laufzeit einer Suchanfrage. Beide Abbildungen zeigen ebenfalls die Laufzeit in Millisekunden auf der X-Achse. Abbildung 6.3 zeigt die Länge der eingegebenen Suchanfrage von 4 bis 10 Zeichen auf der X-Achse. Zu sehen ist ein eher unregelmäßiger Verlauf der Graphen. Es ist jeweils ein starker Anstieg der Suchzeit bei Suchanfragen der Länge 5 auf 11,5 Millisekunden und bei Suchanfragen der Länge 10 auf bis zu 8,38 Millisekunden zu erkennen. Diesen Maxima folgt die kontinuierliche Verringerung der Suchzeit bei steigender Länge der Suchanfrage bis zum jeweils nächsten Maximum. Der Verlauf liegt in der Funktionsweise der Suchmaschine, bzw spezifischer in der jeweils gewählten Editierdistanz begründet. Für alle hier vorgestellten Kurven wurde eine Editierdistanz von $Schlsselwortlnge/5$ gewählt. Dadurch wird die Anzahl der zu betrachtenden und der aktiven Knoten aufgrund der gestiegenen Editierdistanz bei Suchanfragen der Länge 5 und 10 erhöht, welche der Grund für die steigende Laufzeit sind.

Abbildung 6.4 zeigt auf der X-Achse die Anzahl der Worte innerhalb der Suchanfrage. In diesem Szenario wurden mehrere Schlüsselwörter beliebiger Länge zufällig ausgewählt extrahiert und als Suchanfrage genutzt. Die Laufzeit der Kandidatenermittlung steigt linear zur

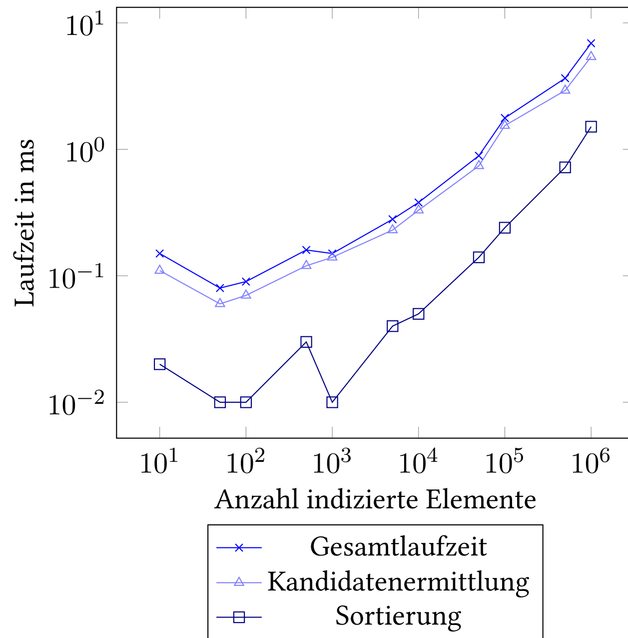


Abbildung 6.2: Suchanfrage - Laufzeit, Indexgröße variabel
Laufzeit einer Suchanfrage bei einem Suchterm p, $|p| = 6$ und variabler Indexgröße.

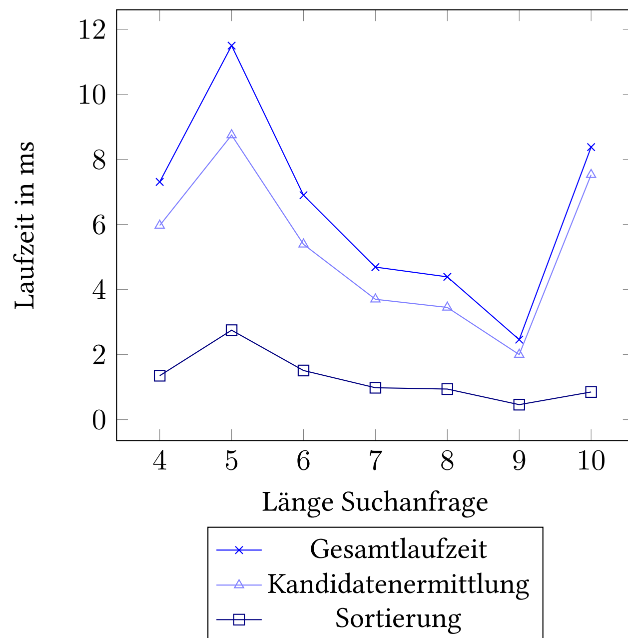


Abbildung 6.3: Suchanfrage - Laufzeit, Suchtermlänge variabel
Laufzeit einer Suchanfrage bei einem Index mit 1.000.000 Elementen und einem Suchterm p mit variabler Länge.

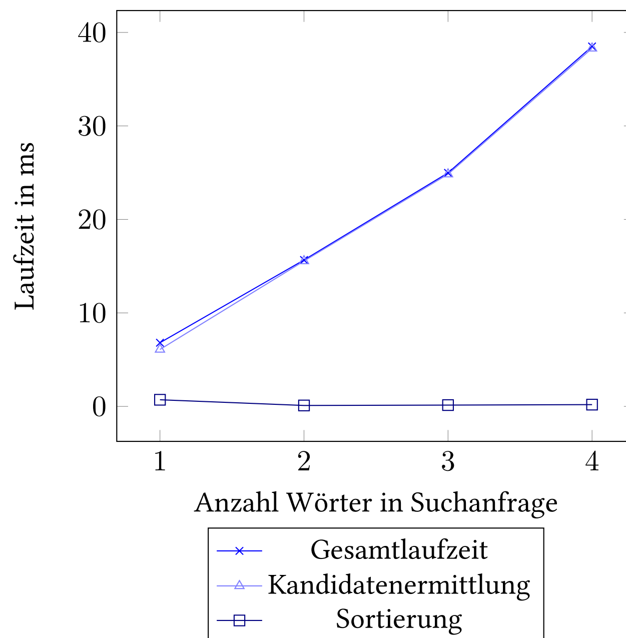


Abbildung 6.4: Suchanfrage - Laufzeit, Anzahl Suchterme variabel
Laufzeit einer Suchanfrage bei einem Index mit 1.000.000 Elementen und variabler Anzahl der Suchterme.

Anzahl der Suchterme. Der Grund darin liegt, dass für jedes Schlüsselwort eine weitere Suche im Trie nach Vervollständigungen erfolgt und diese Ergebnisse vereinigt werden müssen. Es fällt auf, dass die benötigte Zeit für die Sortierung der Elemente allerdings deutlich geringer ist. Der Grund liegt darin, dass durch das Bilden der Schnittmenge der einzelnen Kandidatenlisten die Anzahl der Kandidaten stark reduziert wird.

6.2.2 Vergleich von NGram- und META-Suchmaschine

Wir vergleichen das NGram-basierte und das META-basierte Verfahren in Bezug auf Indizierungszeit, Größe des Recalls und Laufzeit für verschiedene Suchanfragen. Tabelle 6.1 zeigt die durchschnittliche Laufzeit zur Erstellung des Index auf Basis des vollständigen HVV-Datensatzes nach 1000 Durchläufen. Die META Suchmaschine benötigt mit etwa 2 Sekunden nur knapp über ein Drittel der Zeit des N-Gram basierten Verfahrens zur Erstellung des Index.

Abbildung 6.5 zeigt das Laufzeitverhalten der bisherigen und der hier vorgestellten Suchmaschine für Anfragen verschiedener Länge. Für jede Suchmaschine wurde jeweils die Gesamtlaufzeit vom Absetzen der Suchanfrage bis zur Rückgabe der sortierten Kandidatenliste, sowie die Laufzeit der Kandidatenermittlung und die Laufzeit für die Sortierung der Ergebnisse

Algorithmus	Laufzeit in ms
META	2030,96
NGram	5453,32

Tabelle 6.1: Laufzeit der Indexerstellung für den HVV-Datensatz

gemessen. Die roten Graphen repräsentieren das NGram basierte Verfahren, die blauen die METASuchmaschine. Auf der X-Achse ist die Länge der Suchanfrage und auf der Y-Achse die Laufzeit in Millisekunden zu erkennen. die NGram basierte Suchmaschine, im Folgendenen nur noch als NGram bezeichnet, beginnt bei einer Laufzeit von etwa 20 Millisekunden bei einer Termlänge von 4 und weist eine lineare Reduktion der Laufzeit bis zu 1,46 Millisekunden bei Suchanfragen der Länge 10 auf. Die Sortierung der Elemente stellt für NGram einen äußerst geringen Anteil der Gesamtlaufzeit von maximal 0.27 Millisekunden bei Wortlänge 4 dar. META ist mit 1,84 Millisekunden für Anfragen der Länge 4 um den Faktor 10 Schneller als NGram. Bis Suchanfragen der Länge 6 ändert sich die Laufzeit mit 1,68 Millisekunden eher gering, ehe sie dann auf 0.26ms bei Wortlänge 9 abfällt und erst mit der erneuten Steigerung der Editierdistanz bei Wortlänge 10 sich mit 0,88 Millisekunden der Laufzeit von NGram wieder etwas annähernd, alledings auch hier immernoch etwa um nicht ganz den Faktor 2 schneller ist. Es zeigt sich, dass META insbesondere für kurze Anfragen der Länge 4 um einen Faktor 10 schneller Antworten zurückgibt, als die NGram basierte Lösung.deutlich schneller. Erst bei Anfragen der Länge 10 nähern sich die Laufzeiten der beiden Verfahren mit 1.46ms zu 0.82ms an. Dies liegt daran, dass NGram mit einer festen, gewichteten Editierdistanz von 18 arbeitet, während die METASearchEngine die Editierdistanz abhängig von der Länge der Schlüsselwörter variiert. Auch zu erkennen ist, dass die Sortierung der Kandidaten in META einen größeren Anteil an der Gesamtlaufzeit nimmt als bei NGram.

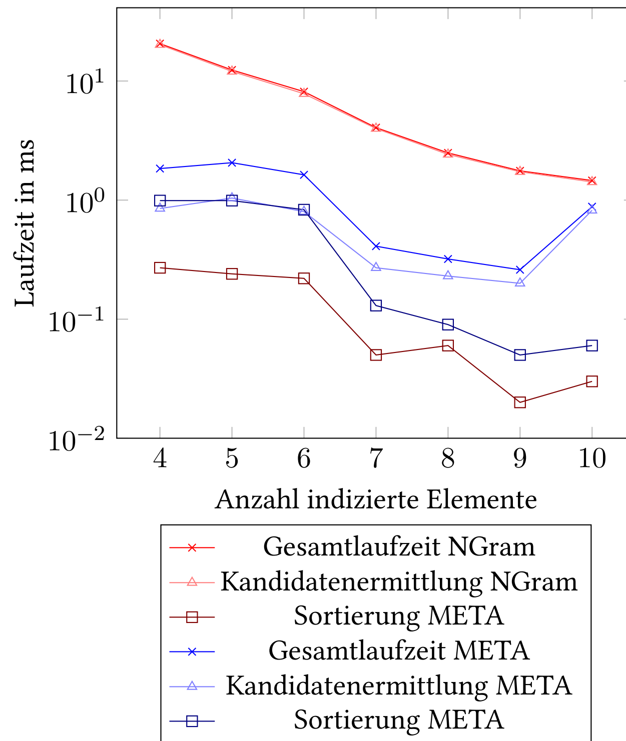


Abbildung 6.5: Suchanfrage - Laufzeit auf dem HVV-Datensatz
Vergleich der Laufzeit beider Verfahren auf dem HVV-Datensatz.
Es zeigt sich, dass der entwickelte Prototyp zum Teil eine deutlich verbesserte Laufzeit als NGram aufweist und auch bei einem steigenden Datensatz noch in einem akzeptablen Rahmen skaliert.

7 Zusammenfassung und Ausblick

In diesem Kapitel wird der Inhalt der Arbeit sowie die wesentlichen Ergebnisse noch einmal zusammengefasst. Abschließend wird ein Ausblick darauf gegeben, welche Verbesserungen an dem entwickelten Prototypen noch vorgenommen werden können und in welchen Bereichen eventuell weitere Forschungsansätze bestehen.

7.1 Zusammenfassung

In dieser Arbeit wurde eine Suchmaschine zur fehlertoleranten Suche von Orten (oder auch anderen Daten) entwickelt und prototypisch implementiert. In Kapitel 2 wurden dafür zuerst grundlegende Begrife und Datenstrukturen beschrieben. In Kapitel 3 sind die vorhandenen Daten sowie die Anforderungen an das System festgehalten. Kapitel 4 hat schließlich vier verschiedene Verfahren aus der Wissenschaft zur Lösung der gegebenen Problemstellung näher beleuchtet. In Kapitel 5 wurde darauf aufbauend ein Konzept für eine Suchmaschine zur fehlertoleranten Suche von Orten innerhalb der Geofox-Anwendung entwickelt und die Implementation näher beschrieben, während sich Kapitel 6 mit der Evaluation des Prototypen insbesondere im Vergleich mit der zuvor befindlichen Suchmaschine befasst.

Es zeigte sich, dass der Prototyp auch für größere Datenmengen, beispielsweise bei einer Ausweitung für das Gesamtdeutsche Verkehrsnetz, ein gutes Laufzeitverhalten aufweist. Auch im Vergleich zum vorherigen Suchalgorithmus ist die Performance des Prototypen positiv zu bewerten, da die Reaktionszeiten des Prototypen bei Suchanfragen teils um einen Faktor 10 geringer sind.

7.2 Ausblick

Die hier vorgestellte Suchmaschine hat noch Potenzial für einige Verbesserungen. So war es mir aus Zeitgründen nicht möglich, das im Abschnitt Abschnitt 4.3 vorgestellte Verfahren zur Bildung der Schnittmenge von Listen, sowie das Caching von Ergebnissen zu vorherigen Eingaben einzubauen. Auch von Interesse könnte eine Erweiterung um die Berechnung einer

bestimmten Anzahl der besten Kandidaten sein. Der Gedanke dahinter ist, dass auf der Nutzeroberfläche ohnehin nur eine begrenzte Anzahl von Vorschlägen angezeigt werden kann und die Editier-Distanz nur dann erhöht wird, wenn nicht mehr ausreichend Vervollständigungen zur Verfügung stehen um die Liste mit einer gewünschten Anzahl von Einträgen zu füllen. Dies kann zum einen die Berechnungsdauer von Suchanfragen von eher kurzen Anfragen verringern, da die Editierdistanz stets minimal bleibt und zugleich die Trefferwahrscheinlichkeit erhöhen, da selbst bei langen Wörtern die Editierdistanz solange gesteigert wird, bis wieder ausreichend Vorschläge zur Verfügung stehen. Deng u. a. (2016) stellt ein Verfahren vor, wie sich der META-Algorithmus mit verhältnismäßig geringem Aufwand für Top-K-Suchanfragen modifizieren lässt. Ein weiterer Faktor stellt die Sortierung dar, welche bisher eher simpel gehalten und nicht näher betrachtet wurde. Durch weitere Untersuchungen lassen sich vermutlich geeignetere Gewichtungen zur Sortierung finden, welche die Relevanz von Einträgen für den Nutzer eventuell besser repräsentieren und so die Qualität des Recalls steigern.

Literaturverzeichnis

[Boytsov 2011] Boytsov, Leonid: Indexing Methods for Approximate Dictionary Searching: Comparative Analysis. In: *J. Exp. Algorithmics* 16 (2011), Mai, S. 1.1:1.1–1.1:1.91. – URL http://doi.acm.org/10.1145/1963190.1963191. – ISSN 1084-6654

[Celikik und Bast 2009] Celikik, Marjan ; Bast, Holger: Fast Error-tolerant Search on Very Large Texts. In: *Proceedings of the 2009 ACM Symposium on Applied Computing*. New York, NY, USA : ACM, 2009 (SAC '09), S. 1724–1731. – URL http://doi.acm.org/10.1145/1529282.1529669. – ISBN 978-1-60558-166-8

[Chaudhuri und Kaushik 2009] Chaudhuri, Surajit ; Kaushik, Raghav: Extending Autocompletion to Tolerate Errors. In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM, 2009 (SIGMOD '09), S. 707–718. – URL http://doi.acm.org/10.1145/1559845.1559919. – ISBN 978-1-60558-551-2

[Deng u. a. 2016] Deng, Dong ; Li, Guoliang ; Wen, He ; Jagadish, H. V. ; Feng, Jianhua: META: An Efficient Matching-based Method for Error-tolerant Autocompletion. In: *Proc. VLDB Endow.* 9 (2016), Juni, Nr. 10, S. 828–839. – URL http://dx.doi.org/10.14778/2977797.2977808. – ISSN 2150-8097

[Garber 2013] Garber, Megan ; TheAtlantic.com (Hrsg.): *How Google's Autocomplete was ...Created / Invented / Born*. August 2013. – URL https://www.theatlantic.com/technology/archive/2013/08/how-googles-autocomplete-was-created-invented-born/278991/. – Zugriffsdatum: 27.03.2018

[Knuth 1973] Knuth, Donald E.: *The Art of Computer Programming, Volume 3: Sorting and Searching*. Reading, MA, USA : Addison-Wesley Publishing Co., Inc., 1973. – ISBN 0-201-03803-X

[Li u. a. 2011] Li, Guoliang ; Ji, Shengyue ; Li, Chen ; Feng, Jianhua: Efficient Fuzzy Full-text Type-ahead Search. In: *The VLDB Journal* 20 (2011), August, Nr. 4, S. 617–640. – URL http://dx.doi.org/10.1007/s00778-011-0218-x. – ISSN 1066-8888

[Miller 1968] Miller, Robert B.: Response Time in Man-computer Conversational Transactions. In: *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I.* New York, NY, USA : ACM, 1968 (AFIPS '68 (Fall, part I)), S. 267–277. – URL http://doi.acm.org/10.1145/1476589.1476628

[Navarro u. a. 2000] Navarro, Gonzalo ; Baeza-yates, Ricardo ; Sutinen, Erkki ; Tarhio, Jorma: Indexing Methods for Approximate String Matching. In: *IEEE Data Engineering Bulletin* 24 (2000), S. 2001

[Xiao u. a. 2013] Xiao, Chuan ; Qin, Jianbin ; Wang, Wei ; Ishikawa, Yoshiharu ; Tsuda, Koji ; Sadakane, Kunihiko: Efficient Error-tolerant Query Autocompletion. In: *Proc. VLDB Endow.* 6 (2013), April, Nr. 6, S. 373–384. – URL http://dx.doi.org/10.14778/2536336.2536339. – ISSN 2150-8097

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 12. Oktober 2018 Ralf von der Reith