Bachelor Thesis

Finding Names on University Web Pages

Patrick Bumüller

Gutachterin: Prof. Dr. Hannah Bast
Betreuer: Niklas Schnelle

Albert-Ludwigs-Universität Freiburg
Technische Fakultät
Institut für Informatik
Professur für Algorithmen und Datenstrukturen

01. März 2019

Bearbeitungszeit

03. 12. 2018 – 03. 03. 2019

Gutachterin

Prof. Dr. Hannah Bast

Betreuer

Niklas Schnelle

Declaration

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I hereby also declare, that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Place, Date	Signature

Abstract

An der Universität Freiburg arbeiten 6.736 Personen, von denen viele eine eigene Webseite haben [1]. Die gesamte Anzahl an Webseiten der Universität übersteigt 300.000 Seiten (5.1.4). Beginnend bei der Startwebseite einer Universität einen Mitarbeiter zu finden ist umständlich. Ohne Vorwissen über die Struktur der Webseiten werden viele Klicks benötigt. In dieser Arbeit wird ein System vorgestellt, das für Bildungseinrichtungen die relevanten Webseiten für Mitarbeiter und andere Personen findet und bewertet. Es implementiert ein automatisches Programm, welches Hyperlinks in Webseiten folgen und Namen im sichtbaren Webseitentext erkennen kann. Es konnten in Stichproben durchschnittlich über 90 % der Mitarbeiterwebseiten gefunden werden (5.1.4).

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Um als Studierender oder Schüler*in an einer Bildungseinrichtung die Sprechstunden eines Mitarbeiters zu finden, eignet sich die Nutzung einer Websuchmaschine. Das Navigieren über die Webseiten einer Universität kann viele Klicks und Vorwissen über den Aufbau dieser benötigen. Bei der Erstellung von Stichproben für dieses System waren oft mehr als zehn, mindestens aber fünf Klicks notwendig, bevor die gewünschte Mitarbeiterwebseite erreicht wurde. Um die Anzahl der Klicks und den damit verbundenen Zeitaufwand zu verringern, eignet sich die Nutzung des hier in dieser Thesis untersuchten Systems. Dieses System bietet eine Möglichkeit, den oben beschriebenen Prozess zu beschleunigen und zu vereinfachen.

Webseiten von Mitarbeitern haben häufig keine eindeutigen Dateinamen und -formate, liegen tief in Seitenstrukturen und unterscheiden sich in ihrem Design. Zudem unterliegt die Erreichbarkeit starken Schwankungen durch das Internet.

Das in dieser Arbeit beschriebene System implementiert einen Webcrawler wie in 1.1.1 beschrieben, der den Webseitenbereich einer Bildungseinrichtung nicht verlässt. Dafür wird der netloc der URL, siehe Abschnitt 1.1.4, einer Webseite auf eine aus der Startseite extrahierte Hauptdomain überprüft. Der Crawler sammelt mit Hilfe eines

HTML Parsers den sichtbaren Text der besuchten Webseiten und speichert diesen pro Webseite in eine Datei. Das System hält sich dabei an die path Restriktionen aus vorhandenen *robots.txt* Dateien, wie in Abschnitt 1.1.2 beschrieben. Angegebene Abfrageraten und -verzögerungen werden nicht berücksichtigt. Um nur HTML Inhalte abzufragen, verwendet das System diverse Überprüfungen wie Dateiendungs- und Dateiheaderchecks. Um unnötige Downloads zu vermeiden, versucht der Crawler Webseiten nur einmal zu besuchen. Hierfür werden während der Laufzeit alle bereits besuchten Seiten in Form von gekürzten absoluten URLs gespeichert.

Ein zweites Teilsystem versucht mit einem regelbasierten Algorithmus natürliche Namen im Seitentext zu finden. Dafür verwendet der Algorithmus vorverarbeitete Listen von Teilnamen. Diese Namensteile werden aus Namen von WikiData Items, siehe 1.1.1, die zu einer Person gehören, erstellt. Der Seitentext wird wortweise gelesen. Wörter, die als bestimmter Namensteil erkannt wurden, verändern den Zustand des Algorithmus. Sobald die Zustandsabfolge einem definierten Muster für einen natürlichen Namen entspricht, werden die gelesenen Wörter zu einem Namen zusammengesetzt und für weitere Schritte normalisiert. Die Normalisierung besteht aus Zeichenersetzungen.

Nach der Untersuchung aller Webseitentexte nach natürlichen Namen, wird für jedes Namen-Webseitenpaar ein Score berechnet. Diese Implementierung verwendet Worthäufigkeitsmaße als Score. Diese sollen die Relevanz der Webseite für diesen Namen widerspiegeln.

Im letzten Schritt wird die Liste aller Webseiten pro Namen anhand des berechneten Score nach Relevanz absteigend sortiert.

Das System enthält auch ein Script, mit dem aus WikiData die offiziellen Webseiten von Bildungseinrichtungen abgefragt werden können. Diese Webseiten können als Startseiten für den Webcrawler verwendet werden.

Durchschnittlich $94,95$ % der Mitarbeiterseiten aus repräsentativen Stichproben wurden in Crawls über verschiedene Bildungseinrichtungen gefunden (5.1).

1.1 Definitionen

Nachfolgende Begriffe und Systeme werden in dieser Arbeit verwendet. Zudem wird eine einheitliche Benennung definiert.

1.1.1 Webcrawler

In dieser Arbeit werden Teile des Systems als Webcrawler oder kurz Crawler bezeichnet. Ein solches Programm oder Teilprogramm kann automatisch Links in Webseiten finden und folgen [2]. Dies kann zu zusätzlicher Last für den Seitenbetreiber oder unerwünschtes Verhalten sorgen, welches im Abschnitt 1.1.2 weiter ausgeführt wird.

1.1.2 robots.txt

In "*robots.txt*" benannte Textdateien können Betreiber von Webseiten speziellen Besuchern wie Webcrawlern mitteilen, welche Bereiche der Seite sie besuchen dürfen [3]. Zusätzlich können diese Informationen zur Abfragerate und -verzögerung beinhalten. Für die Einschränkung der Bereiche kann eine "*robots.txt*" Datei beliebig viele Einträge mit erlaubten und verbotenen URL path Präfixen enthalten.
Das Einhalten der Vorgaben aus "*robots.txt*" Dateien ist nicht verpflichtend, gehört aber zur Netiquette im Internet.

1.1.3 WikiData

WikiData ist eine quelloffene und kostenlose Knowledgebase. Sie bietet Schnittstellen für Menschen und Maschinen und dient als zentraler Sammelpunkt für strukturierte Daten ihrer Schwesterprojekte der Wikimedia Foundation [4]. Wikipedia ist ein solches Schwesterprojekt.
Als Knowledgebase bezeichnet man eine Wissenssammlung. Sie kann verschiedene

Bereiche abdecken und strukturierte sowie unstrukturierte Informationen enthalten. Erstere haben definierte Gemeinsamkeiten. Zum Beispiel können sie gleiche Vererbungshierarchien, Ähnlichkeiten oder relationale Zusammenhänge haben.
Das Resource Description Framework, kurz RDF, ist eine Spezifikation, um Informationen in Form von Web Ressourcen zu modellieren [5]. Web Ressourcen beschreiben dabei Daten und eindeutige Identifikatoren, die auch Uniform Resource Identifier (URI) genannt werden. In RDF werden Informationen als Triple modelliert. Jedes dieser Triple hat die Form

Subjekt Pädikat Objekt

Alle Elemente des Triples sind Web Ressourcen. Das Subjekt ist die zu beschreibende Information. Das Prädikat definiert die Relation und das Objekt enthält die Zielinformation.

Abbildung 1: Diese Abbildung zeigt, wie strukturierte Daten als RDF-Triples dargestellt werden können.

Abbildung 1 zeigt vereinfacht ein solches Triple.
In WikiData heißen strukturierte Informationen Items oder Entitäten. Sowohl Subjekt, Prädikat als auch Objekt können WikiData Items sein.

1.1.4 URL

Ein Uniform Resource Locator (URL) ist eine Zeichenkette, die verwendet werden kann, um Ressourcen über das Internet abzufragen [6]. Sie sind nach Definition spezielle URIs.
Das für dieses System verwendet Python urllib Modul implementiert eine in RFC-1808 eingeführte Erweiterung durch relative URLs [7]. In dieser Arbeit werden Teile einer

URL wie in RFC-1808 benannt. Eine URL besteht laut RFC-1808 aus

`<scheme>://<netloc><path><params><query><fragments>`

`<scheme>` bezeichnet das verwendete Protokoll, z. B. `http`. Der `<netloc>` oder auch authority genannt enthält optionale Informationen wie den Host, Userinfo und Port. Der `<path>` gibt den Pfad zur Ressource an. Alle anderen Bestandteile `<params>`, `<query>` und `<fragments>` werden von diesem System nicht verwendet.
Ist der netloc keine IPv4 oder IPv6 Adresse, darf diese Zeichenkette maximal 253, plus Userinfo und Port, Zeichen lang sein [8].

2 Verwandte Arbeiten

Die in [9] ersten beschrieben Webcrawler verwenden Graphen basierte Strategien, um alle Seiten ohne Beachten des Inhalts zu crawlen. Verwendete Strategien können Tiefen- oder Breitensuchen sein. Diese benötigen eine Startmenge von welchen aus entsprechende Suchen beginnen. Die Startmenge wird auch Seed genannt. Je nach Verwendungszweck speichern die Crawler nur Teile oder die gesamte Seite. Sie besuchen Seiten nicht zwangsläufig nur einmal. Verwendet werden diese bei der Erstellung von Indexen für Suchmaschinen.
Auch das in dieser Arbeit verwendete System nutzt eine einfache Suchstrategie in Form einer limitierten Breitensuche. Daher werden Seiten nur bis zu einer angegebenen Tiefe gecrawlt. Weiterhin wird jede Webseite nur ein einziges Mal besucht.

Das in [10] als Focus Cralwer beschriebene System verwendet komplexere Strategien, um Seiten für das weitere Crawlen auszuwählen. Es nutzt Ansätze, die den Textinhalt einer Seite, die Relevanz und deren Beziehung zu anderen verlinkten oder auf sie linkenden Seiten berücksichtigt. Dadurch kann der Crawl auf die für ein Thema relevanten Seiten reduziert werden. Das System priorisiert anhand der Relevanz das Crawlen und besucht Seiten mit vielen Verlinkungen mehrmals.
Im Kontrast dazu verwendet das in dieser Arbeit implementierte System eine einfache, in 3.1.1 beschriebene, Heuristik für einen Fokus auf die aktuelle Bildungseinrichtung. Wie in 3.2 beschrieben, behandelt das System dieser Arbeit die Seitentextanalyse mit

Fokus auf Namen komplett getrennt vom Crawlen selbst.

Das System aus Artikel [11] verwendet eine austauschbare Layer Architektur. Es wird ein Classifier genutzt, um mithilfe von Beispielseiten einen Crawl Fokus auf deren Inhalte zu legen. Als Features werden Worthäufigkeitsmaße verwendet. Die Forscher haben herausgefunden, das Bayes-Klassifikatoren im Vergleich mit Support Vektor Maschinen und neuronalen Netzen am schlechtesten als fokusgebendes Element arbeiten [11].
Worthäufigkeitsmaße finden auch im System dieser Arbeit Verwendung. Sie werden direkt als Maß für die Relevanz von Webseiten-Namenspaaren verwendet.

3 Das System

Das ganze System unterteilt sich in zwei Teilsysteme. Den Webcrawler (Abschnitt 3.1) und das Finden von natürlichen Namen (Abschnitt 3.2). Der Webcrawler prüft (3.1.2) und lädt Webseiten herunter (3.1.3). Der Namensfinder liest den sichtbaren (3.1.4), vom Webcrawler extrahierten, Webseitentext ein, findet Namen und sortiert Namen-Webseitenpaare nach Relevanz. Beide Teile sind in Python3 implementiert und in einem Ubuntu 18.04 Docker Image getestet.
Das Unterteilen in zwei getrennte Systeme hat den Vorteil, dass Scores für die Namen verwendet werden können, welche die gesamte Dokumentmenge benötigen, wie z. B. bm25 (3.2.2.2). Zudem ist es leichter zwei Teilsysteme getrennt zu entwickeln und zu testen.
Weiterhin gibt es ein Script, welches Bildungseinrichtungen aus der Knowledgebase WikiData extrahieren kann. Es wird in Abschnitt 3.3 beschrieben. URLs aus der von diesem Script erstellten Liste können als Start URLs für den Webcrawler verwendet werden.

3.1 Webcrawler

Der Webcrawler benötigt eine URL als Einstiegspunkt. Aus dieser URL wird die Hauptdomain wie in Abschnitt 3.1.1 beschrieben extrahiert. Diese wird für einen Teil der URL-Überprüfung benötigt. Anschließend wird vom Einstiegspunkt aus eine limitierte Breitensuche auf alle verlinkten URLs angewendet. Alle auf der Startseite

gefundenen URLs sind in Ebene 0, alle ermittelten URLs aller Seiten aus Ebene 0 sind in Ebene 1. Das Gleiche gilt analog für alle weiteren Ebenen. Das System verwendet diese Suche, da sie auf einem engmaschigen Graphen eine gleichmäßigere Topologie beginnend von der Startseite, in Form der oben beschriebenen Ebenen, erstellt. Eine limitierte Tiefensuche könnte, je nach zuerst gewählter Kanten aus der URL Menge, eine Topologie erstellen, die direkte Nachfolger der Ebene 1 erst sehr spät betrachtet. Der Webcrawler arbeitet immer eine Ebene nach der anderen ab. Im Speicher werden immer nur die aktuelle und nächste Ebene gehalten. Um Dopplungen zu vermeiden werden hierfür Set-Datenstrukturen verwendet. Jede gefundene URL wird vor dem Hinzufügen in die nächste Ebene wie in Abschnitt 3.1.2 beschrieben überprüft. Für einen Teil der Überprüfung und das Ermitteln der verlinkten URLs muss die Seite heruntergeladen (3.1.3) und geparst (3.1.4) werden. Jede geparste Seite wird anschließend wie in Abschnitt 3.1.5 beschrieben auf dem Dateisystem gespeichert. Das hat den Vorteil nicht alle Seiten im Speicher halten zu müssen, einen Web crawl schnell abbrechen und die heruntergeladenen Webseiten auch für andere Zwecke nutzen zu können.

3.1.1 Hauptdomain extrahieren

Um die Hauptdomain aus der Einstiegspunkt URL zu extrahieren wird nur der netloc Teil der URL betrachtet. Dieser wird in die einzelnen, durch '.' separierten Domains getrennt und in einer geordneten Teildomain Liste gespeichert. Der Webcrawler lädt beim Starten einmalig eine aktuelle Liste aller Toplevel Domains von der Internet Assigned Numbers Authority herunter und speichert diese in einer Set-Datenstruktur [12]. Beginnend vom letzten Element der Teildomainliste wird rückwärts über diese iteriert und geprüft, ob die aktuell betrachtete Teildomain im Toplevel Domain Set enthalten ist. Die erste Teildomain, welche nicht im Toplevel Domain Set ist, wird als Hauptdomain gesetzt. Diese wird bei einer URL Prüfung verwendet, um den Bereich der Webseiten einer Einrichtung nicht zu verlassen. Die Hauptdomain auf diese Art

zu extrahieren ist mit einem einmaligen Download und der maximal 253 (maximale netloc Länge) Set membership Tests günstig und funktioniert auch, wenn beliebig viele Subdomains der Hauptdomain mit angegeben sind.
Beispielsweise ist *uni-freiburg* die Hauptdomain für die URL *http://www.informatik.uni-freiburg.de/* oder *osaka-sandai* die von *http://www.ojc.osaka-sandai.ac.jp/*.

3.1.2 URL Prüfung

Die URL Prüfungen des Webcrawler haben den Zweck bestimmte Seiten zu ignorieren. Alle URLs die auf nicht HTML Inhalte verweisen, die bereits besucht wurden, außerhalb der Hauptdomain liegen oder vom Besitzer als nicht zu crawlen gekennzeichnet (1.1.2) sind, sollen nicht verarbeitet werden.
Die Überprüfungen lassen sich grob in URL Prüfungen mit und ohne Webseiten-Download unterteilen. Diese Unterteilung hat den Vorteil, dass einige Seiten bereits vor dem langsamen Download aussortiert werden können.

3.1.2.1 URL Prüfung ohne Download

Die URL Prüfungen ohne Download sind einfache Checks auf dem URL String selbst. An dieser Stelle wird auch die *robots.txt* der entsprechenden Subdomain der URL überprüft. Diese muss beim ersten Auftreten der Subdomain heruntergeladen werden. Die folgenden Prüfungen werden in dieser Reihenfolge vorgenommen. Beim Fehlschlagen eines Checks wird die URL verworfen.

- Zuerst wird das URL scheme überprüft. Es sind nur *http* und *https* erlaubt. Diese sind die beiden geläufigsten Protokolle für HTML Inhalte auf Webseiten.

- Anschließend wird der URL netloc nach '.' getrennt und überprüft, ob die Hauptdomain gleich mindestens einem Teilnetloc ist. Dadurch wird gewährleistet, dass der Bereich der Webseite einer Einrichtung nicht verlassen wird.

- Als nächstes wird überprüft, ob die URL bereits besucht wurde, indem ein membership Test auf einer Set-Datenstruktur mit bereits besuchten URL Strings vorgenommen wird. Eine Set membership Test mit amortisiertem $\mathcal{O}(1)$ ist effizienter als ein solcher für Listen Datenstrukturen mit $\mathcal{O}(n)$ [13].

- Danach wird der path der URL auf eine erlaubte Dateiendung getestet. Erlaubt ist eine Auswahl an Dateiendungen die auf HTML Inhalte schließen lassen wie beispielsweise *.htm*, *.php4* oder *.jsp*. Diese sind in einer Set-Datenstruktur gespeichert. Weiterhin sind nicht vorhandene Dateiendungen erlaubt. Eine Whitelist hierfür ist kürzer als eine Blacklist mit verbotenen Endungen.

- Zuletzt wird überprüft, ob für das netloc bereits eine *robots.txt* Datei heruntergeladen und geparst wurde. Wenn nicht, wird dies gemacht und der Parse in einer Dictionary Datenstruktur gespeichert. Anschließend oder wenn bereits vorhanden wird überprüft, ob die im path der URL spezifizierte Ressource als erlaubt markiert ist. Die Implementierung nutzt einen Python RobotFileParser (urllib.robotparse.RobotFileParser). Beim Download der *robots.txt* Datei wird eine Anfrage mit dem User Agent String des Systems (3.1.3) gestellt. Weiterhin werden, ähnlich wie im Abschnitt 3.1.3 beschrieben rekursiv Wiederholungen versucht, wenn verschiedene Exceptions auftreten. Bei einem HTTPError ist das Verhalten gleich dem in [14]. HTTP-Codes 401 und 403 verbieten alle Ressourcen, sonstige Codes im Bereich 400 bis 499, inklusive der Grenzen, erlauben alle Ressourcen. Bei einem SSLError wird einmalig eine Wiederholung mit einem unverifiziertem SSL Kontext versucht, ansonsten werden alle Ressourcen als verboten angesehen. Bei einem IncompleteRead wird eine Wiederholung mit den gleichen Parametern versucht. Für einen OSError oder einer sonstigen Exception wird keine Wiederholung versucht und alle Ressourcen werden als verboten gesetzt. Maximal drei Wiederholungen können vorgenommen werden, bevor alle Ressourcen als verboten angesehen, werden.
  Das Wiederholungsverhalten hat sich in dieser Weise auf realen Seiten bewährt.

Es verbietet im Zweifel alle Ressourcen, was bei Fehlern dem Seitenbetreiber zugutekommt. Um auch selbst signierte und nicht vertraute Zertifikate zu erlauben und da *robots.txt* Textdateien nicht sehr sicherheitskritisch sind, wird einmalig eine Anfrage mit einem nicht verifiziertem SSL Kontext versucht.
Das System ist auch robust gegenüber falschen oder nicht vorhandenen Charsets. Zuerst wird versucht das Charset aus dem Header der Response zu verwenden. Wenn das Charset unbekannt oder nicht vorhanden ist, wird *utf-8* verwendet. Schlägt das Decodieren der Response mit dem Charset fehl, werden alle Ressourcen als verboten angesehen. *utf-8* bietet sich als Standard an, da über 90 % der Webseiten diese Kodierung verwenden [15].

3.1.2.2 URL Prüfung mit Download

Die URL Prüfungen mit Download sind Überprüfungen des Dateityps. Bei der ersten Überprüfung wird untersucht, ob der content-type der Webseite ein Text HTML Typ ist. Falls dies zutrifft gilt die Webseite als Verwendbar. Im Falle eines nur Texttype content-type wird zusätzlich nach einer Doctype HTML Deklaration gesucht. Falls ein HTML Doctype gefunden wurde, wird die Seite weiter verwendet. Ist der content-type kein Texttype, die Doctype Deklaration falsche oder nicht vorhanden, wird die Webseite verworfen.
Durch diese beiden Checks wird zusätzlich zum Dateiendungscheck sichergestellt, dass nur Seiten mit HTML Inhalt weiter verwendet werden.

3.1.3 Download der Webseite

Der Download der Seite wird mit der Python urllib.request.urlopen Methode realisiert. Zusätzlich wird ein eigener User Agent String und ein durch Rekursion realisiertes Wiederholungsverhalten verwendet.

- Ein User Agent String wird verwendet, um Komponenten eines anfragenden Clients dem Server bekanntzumachen. Er kann aus mehreren Komponenten bestehen, wobei jede dieser die Form *Name/Version (Kommentar)* hat [16]. Der im System verwendete User Agent String ist
  `"Mozilla/5.0 Python-urllib/3.6 ScientistFinder/1.0 (Project page: ad-wiki.informatik.uni-freiburg.de/teaching/BachelorAndMasterProjectsAndTheses/ScientistFinder)"`.
  Der erste Teil `Mozilla/5.0` sagt aus, dass die Seite für ein Mozilla kompatibles System dargestellt werden soll. Dieses Token ist bei vielen Webbrowsern Standard [17]. Aus historischen Kompatibilitätsgründen wurden einige Webseiten anders oder gar nicht an Webbrowser ohne diese Komponente ausgeliefert [18]. Daher verwendet auch dieses System diese User Agent Komponente.
  Da das System die Python urllib Implementierung verwendet, ist auch dieser im User Agent String mit entsprechender Version vorhanden.
  Die letzte Komponente beschreibt das System selbst und verlinkt im entsprechenden Kommentarfeld auf die Projekt Webseite, um volle Transparenz beim Crawlen zu gewährleisten.

- Da beim Download der Seite aus dem Internet viele Fehler auftreten können, verwendet das System ein rekursives Wiederholungsverhalten. Dabei wird maximal fünf mal versucht die gleiche Seite herunterzuladen. Zwischen jedem Versuch wird ein *sleep* von 2^{retry} Sekunden, mit $0 \leq retry < 5$, durchgeführt. Dieses Verhalten soll die Last für den Server verringern. Manche Fehler sorgen für ein Abbrechen, ohne weitere Wiederholungen zu versuchen und verwerfen die Seite. Das System fängt HTTPError ab. Diese können verschiedene Codes mit verschiedenen Bedeutungen haben. Das System versucht eine Wiederholung, wenn ein Code zwischen 500 und 599, inklusive der Grenzen, vom Server zurückgegeben wird. Diese Fehlercodes signalisieren üblicherweise einen Serverfehler [19]. Da Serverfehler oft nur temporär sind, wird beim Auftreten dieser mit

beschriebener Verzögerung eine Wiederholung versucht.
Weiterhin werden SSL Error und SSL Verification Error abgefangen. Zuerst wird bei einem solchen Fehler eine Wiederholung mit einem anderen Zertifikatbündel von certifi versucht [20]. Das kann besser funktionieren, wenn auf dem Computer der dieses System ausführt keine oder wenige Zertifikate installiert sind. Beim zweiten Abfangen eines solchen Fehlers wird ein Download mit einem unverifiziertem SSL Kontext versucht. Da das System mit nicht sicherheitsrelevanten Daten arbeitet und zur Unterstützung selbst signierter Zertifikate, wird genau einmal eine Wiederholung versucht. Bei weiteren auftretenden Fehlern dieser Art wird keine Wiederholung versucht.
Beim Anfragen der Webseite wird ein Timeout von 20 Sekunden verwendet. Benötigt eine Anfrage länger, wird ein Timeout Error ausgelöst. Ein solcher Fehler führt zum Erhöhen des aktuellen Timeouts um zehn Sekunden und einem Wiederholungsversuch mit dem neuen Timeout. Durch dieses Verhalten können auch Antworten von stark ausgelasteten Servern beim Crawler ankommen.
URLError und OSError werden ebenso abgefangen. Sie können verschiedene Ursachen haben wie beispielsweise missglückte DNS Anfragen oder Fehler bei einem Handshake. Für alle URLError wird eine Wiederholung durchgeführt. Für OSError, die vorher nicht bereits von anderen spezifischen Exceptions abgefangen wurden, wie einem URLError, wird keine Wiederholung versucht.
Beim Lesen der Webseite kann es vorkommen, dass ein Teil der Seite fehlt. Dadurch wird eine sogenannte IncompleteRead Exception ausgelöst. Der Fehler kann meistens durch einen Wiederholungsversuch mit den gleichen Parametern gelöst werden.
Alle sonstigen Fehler, die keine KeyboardInterrupt sind und noch nicht bereits behandelt wurden führen zu einem Abbruch und keinem weiteren Wiederholungsversuch.

3.1.4 Parsen der Webseite

Für das Parsen der HTML Seite verwendet das System ein Modul namens BeautifulSoup [21]. Dieses Modul erstellt, wie ein Webbrowser, einen kompletten Parse der HTML Seite. BeautifulSoup bietet die Möglichkeit in diesem Parse hierarchisch zwischen den einzelnen Knoten oder in HTML auch Tags genannten Elementen zu navigieren.

Das System verwendet dieses Modul, da es die eben beschriebenen Navigationsmöglichkeiten zwischen einzelnen Tags bietet und dadurch das Filtern von sichtbarem Text im HTML sehr einfach macht. Zudem können auch direkt alle Anker-Tags abgefragt werden. Diese sind für das Crawlen der verlinkten Seiten nötig.

```
(^http://.*$)|(^https://.*$)|(^/.*$)|(^\..*$)|(^\w+[^:]*$)
```

Listing 3.1: Dieses Listing zeigt einen regulären Ausdruck, der nur absolute, relative, http und https Links matcht.

- Aus der geparsten HTML Seite werden alle Anker-Tag Paare gesucht.

  `<a href="http://www.python.org/">Python</a>` ist ein Beispiel für ein solches öffnend- und schließendes Anker-Tag Paar. Im Webbrowser würde es als eine Verlinkung mit dem Namen *Python* und dem Ziel `http://www.python.org/` dargestellt werden. Das im `href` Attribut beschriebene Ziel muss den regulären Ausdruck aus Listing 3.1 erfüllen und wird dann vom System mithilfe von urllib.parse.urljoin mit der URL der aktuellen Seite zusammengefügt. Wenn eine solche Adresse dann die URL Prüfung ohne Download (Abschnitt 3.1.2.1) besteht, wird sie in das nächste Level der Breitensuche hinzugefügt.

  Durch das Prüfen des `href` Felds mit dem regulären Ausdrucks aus Listing 3.1 kommen nur absolute, relative und URLs mit scheme *http* oder *https* weiter. Das anschließende Zusammenfügen der aktuellen Seite und den gefundenen Links durch urllib.parse.urljoin setzt absolute und relative Ziele richtig zu einer absoluten URL zusammen.

  Wie in Abschnitt 1.1.4 bereits erwähnt werden nur scheme, netloc und path

einer URL behalten, alle anderen URL Komponenten werden verworfen.
Fragmente sind nicht Teil der URL und können daher vernachlässigt werden [22]. Da das System nur statische HTML Inhalte betrachtet, können params und query ebenfalls weggelassen werden.

- Das System arbeitet nur mit dem sichtbaren Text im HTML. Als sichtbar ist dabei alles zu verstehen, was beim Besuchen der Website mit einem Webbrowser von diesem sichtbar für den Nutzer im Hauptfenster angezeigt wird. Das umfasst z. B. Überschriften, Paragrafen oder Aufzählungen. Beispiele für nicht sichtbare Texte sind HTML Kommentare, der Webseitentitel im Fensterrahmen oder Metainformationen. Das Filtern kann mit BeautifulSoup durch das Ignorieren von HTML-Tags mit verbotenen Eltern-Tags und dem Prüfen auf eine Kommentarinstanz geschehen. Alle Kinder der Tags `'style', 'script', 'head', 'title', 'meta', 'document'` und Kommentar-Tags werden nicht verwendet. Bei allen übrigen Tags wird der eingeschlossene Text abgefragt und zeilenweise in einem String gespeichert.
  Die Verwendung von nur sichtbarem Text hat den Vorteil, dass die zu speichernde Information kleiner als die ganze HTML-Seite ist und die Namenserkennung keine HTML-Tags beachten muss.

3.1.5 Speichern der Webseite

Das Crawler Teilsystem speichert den sichtbaren Text, wie in Abschnitt 3.1.4 beschrieben, einer Webseite zeilenweise in eine Datei. Dadurch können diese Texte auch für andere Zwecke verwendet werden und müssen nicht über die gesamte Laufzeit im Speicher gehalten werden.
Das System kann für das Speichern einer Seite einen beliebigen Opener verwenden. Dieser kann über eine Python Konfigurationsdatei für alle Komponenten, die diese Dateien öffnen, geändert werden. Standardgemäß wird ein bz2.open Opener

verwendet [23]. Dieser komprimiert/dekomprimiert Dateien beim Öffnen mit der Standardkonfiguration des bzip2 Algorithmus. Das System verwendet diese Komprimierung, da sie in verschiedenen Benchmarks eine gute Kompressionsrate, eine mittlere Kompressions-/Dekompressionszeit hat und am zweit- bis drittwenigsten Arbeitsspeicher benötigt [24, 25].
Als Dateiname wird die URL der Seite selbst verwendet. Da einige Zeichen in URLs nicht als Dateinamen erlaubt sind, müssen diese durch andere ersetzt oder, falls möglich, maskiert werden. In vielen Dateisystemen, darunter auch ext4 von Ubuntu 18.04, sind "/" Zeichen nicht im Dateinamen erlaubt [26]. Das System ersetzt diese durch den Fragment-Identifier "#". Dieser kann sicher verwendet werden, da es sich um ein Punktationszeichen handelt, welches nur für Fragmente verwendet wird und diese vom System, wie in Abschnitt 3.1.4 beschrieben, weggelassen werden [27]. Diese und weitere Substitutionen im Dateinamen können durch Anpassen der Python Konfigurationsdatei verändert werden. Da es in Dateisystemen auch eine Beschränkung für die maximale Dateinamenslänge gibt, werden bei zu langen Dateinamen neue Namen der Form `too_long_filename<counter>` gebildet und die unmaskierte URL in der ersten Zeile einer solchen Datei gespeichert. Das Präfix für solche Dateien lässt sich wieder über die Python Konfigurationsdatei ändern.
Die URL als Dateinamen zu verwenden macht das Debugging leicht und beschleunigt die Coverage Evaluation vorhandener URLs in Abschnitt 5.1, da die Dateien selbst nicht dekomprimiert werden müssen.

3.2 Finden von natürlichen Namen

Der Namensfinder benötigt ein Verzeichnis in dem alle Crawlseiten liegen und drei Dateien die Namensteile enthalten. Letztere sind Textdateien die, zeilenweise jeweils getrennt Vor-, Mittel- und Nachnamen enthalten. Das Namensfinder-Teilsystem bietet eine Möglichkeit, diese Dateien aus den Namen von WikiData Entitäten zu erstellen. In Abschnitt 3.2.1 wird dieser Vorgang genauer beschrieben.

Mit diesen Daten versucht das System Namen pro Text einer einzelnen Crawldatei zu finden (3.2.2). In den Texten werden `"\n"` Zeichen, welche einen Zeilenumbruch signalisieren, durch `";"` ersetzt. Dadurch werden unnötig lange Namenstreffer durch die verwendete Heuristik (3.2.2.1) der Statemaschine verringert. `"\n"` Zeichen kommen dabei nicht aus dem HTML, sondern sind eine Folge zeilenweisen Speicherns (3.1.5) des sichtbaren Textes. In Abschnitt 3.2.2.1 wird die Namenserkennung durch die verwendete Statemaschine genauer beschrieben.
Es werden zusätzlich words-, docs- und ein nt-file (3.2.2.4), die als Text/Knowledgebase in QLever verwendet werden können, erstellt [28, 29]. Wenn das System alle Crawlfiles abgearbeitet hat, werden für Namen-Webseitenpaare Scores berechnet (3.2.2.2), um die Relevanz einer Seite für einen Namen auszudrücken. Als Ergebnis erstellt dieses Teilsystem eine Datei wie in Abschnitt 3.2.2.3 beschrieben. Diese enthält normalisierte Namens-Tags und eine sortierte Liste mit allen Webseiten, in welcher dieser Name vorkommt, und deren Scores für den Namen.
Durch eine Normalisierung werden Namen eindeutig. Die für die Normalisierung nötigen Schritte werden in Abschnitt 3.2.2 beschrieben.

3.2.1 Teilnamen Set-Datenstrukturen

Für die Erkennung von Namen in Texten verwendet das System drei Set-Datenstrukturen, die jeweils getrennt Vor-, Mittel- und Nachnamen enthalten. Um diese zu erstellen, fragt das System ein WikiData Backend nach den englischen Labels der Entitäten, die von der Entität *Person* erben ab. Das Listing 3.2 zeigt die entsprechende SPARQL Query dazu. Dabei stehen *wdt:P31* für die Relation mit Präfix *instance_of*, *rdfs:label* für die Relation mit Präfix *Label* und *wd:Q5* für die Entität *Person*.
Die durch die SPARQL Query in Listing 3.2 erhaltenen Strings mit vollen Namen werden nach Leerzeichen und optional `"-"` in Namensteile getrennt. Wenn es nur einen Namensteil gibt, wird dieser als Nachname betrachtet, bei zwei wird der erste als Vor- und der zweite als Nachname gespeichert und bei drei oder mehr

```
SELECT DISTINCT ?name WHERE {
    ?id wdt:P31 wd:Q5 .
    ?id rdfs:label ?name .
    FILTER(lang(?name) = 'en')
} LIMIT 200000
```

Listing 3.2: Dieses Listing zeigt eine Query, die englische Namen von Personen zurück gibt. Es werden maximal 200.000 Ergebnisse abgefragt.

Namensteilen werden wieder der erste und letzte Namensteil als Vor- und Nachname, alle weiteren Teile als Mittelnamen, betrachtet. Wenn alle vollen Namen auf diese Weise getrennt wurden, wird die Mittelnamen Set-Datenstruktur einzigartig vom Vornamen Set gemacht. Dies geschieht durch eine mathematische Mengendifferenz $mittelname = mittelnamen - vornamen$.
Optional können aus den einzelnen Set-Datenstrukturen noch Namensteile, die Zahlen sind oder eine zu kleinen Länge haben, gefiltert werden. Die Filterung von Zahlen und eine Mindestlänge von 1 sind die Standardeinstellungen.
Das System bietet zusätzlich Funktionalitäten die Namens Set-Datenstrukturen zeilenweise jeweils getrennt in Textdateien zu schreiben und aus diesen zu lesen.

3.2.2 Namensfinder

Der vom System pro Crawlfile gelesene Text wird an den Namensfinder übergeben. Dieser ist durch eine Superklasse definiert, von welcher erbende Klassen eine *find_names* Methode implementieren müssen, welche die Namenserkennung implementiert. Eine solche Realisierung bekommt den Text, die drei Sets Datenstrukturen mit Namensteilen und eine optionale Namenslänge als Input. Die Methode muss ein zweistelliges Tuple zurückgeben. Dieses Tuple besteht aus einem Text, in dem Namen durch Digit-Tags der Form `<[0-9]+>` ersetzt wurden und einer Dictionary Datenstruktur, die für jedes Digit-Tag aus dem Text als Key den gefunden vollen

Namen als Value enthält.
Durch die Superklasse werden einige nützliche Methoden bereitgestellt.

- Eine *lookup* Methode, die Set membership Tests mit optionalen Modifikationen durchführt. Unterstütze Optionen sind den abzufragenden Wert komplett in Kleinbuchstaben oder nur den ersten Buchstaben in Großbuchstaben darzustellen. Anhängenden Interpunktionen am abzufragenden Wert zu entfernen und eine Kombination dieser Optionen.

- Eine *get_ words* Methode, die einen Text nach Whitespaces in eine Liste einzelner Worte trennt und diese optional modifiziert. Alle Wörter, welche den regulären Ausdruck `<[0-9]+>` erfüllen werden gefiltert. Wörter dieser Form werden für das interne Tagging von Namen mit Digit-Tags verwendet. Optional können alle Interpunktionen von Wörtern entfernt werden.

- Eine Methode, die hinten anhängende Namensteile, die in einem angegebenen Set vorkommen, entfernt. Zum Beispiel wird so der fast Name "*Christian Edgar von und*" zum Namen "*Christian Edgar*", wenn eine Set-Datenstruktur übergeben wird, welches die Strings "*von*" und "*und*" enthält. Diese Methode wird für hinten anhänge voreilig mit erkannte Mittelnamen, wie im Beispiel, verwendet.

- Eine Methode, die Namen normalisiert und festlegt, welche Namensworte mit dem normalisiertem Namen für das wordsfile (3.2.2.4) markiert werden sollen. Das System normalisiert Namen, indem Leerzeichen durch "_" ersetzt, Präfixtitel entfernt werden und alles in <> Klammern eingeschlossen wird. Zum Beispiel wird "*Herr Dennis Armbruster*" zu "*<Dennis_ Armbruster>*".
  Auch bei der Markierung für das wordsfile werden nur die Präfix-titel nicht mit dem normalisierten Namen markiert (3.2.2.4).

3.2.2.1 Statemaschine Namen Finder

Das System verwendet eine Namensfinder Statemaschine-Implementierung. Diese funktioniert wortweise wie ein Streamparser.

```
<prefix_title>*((<first_name>+)|((<first_name>+)|(<
    first_name>+<last_name>+)|(<first_name>+<middle_name>+<
    last_name>+)|(<prefix_title>+<last_name>+))
```

Listing 3.3: Dieses Listing zeigt einen Ausdruck, der die von der Statemaschine erkannten Namen darstellt.

Mit der Statemaschine können Namen bestehend aus Namenswörtern, wie in Listing 3.3 aufgezeigt, dargestellt werden. Dabei ist `<prefix_title>` ein regulärer Ausdruck und `<first_name>`, `<middle_name>` und `<last_name>` sind Set-Datenstrukturen. Entsprechend müssen Namensworte bei den Set-Datenstrukturen einen membership-Test bestehen und bei einem regulären Ausdruck einen positiven match ergeben.
In Abbildung 2 ist die Statemaschine als Graph dargestellt. INIT ist der Startzustand. Jeder Zustand beschreibt das zuletzt gelesene Wort. Zum Beispiel signalisiert der MIDDLE Zustand, dass das letzte Wort ein Mittelnamenswort war. Wenn ein Wort nicht als Namenswort erkannt wurde, bleibt der Zustand auf INIT. Die Kanten haben zusätzliche Bedingungen, die sich nicht zwangsläufig ausschließen. Die Nummerierung gibt an, in welcher Reihenfolge die Kantenbedingungen geprüft werden sollen. Alle Kanten konsumieren das nächste Wort beim Übergang. Davon ausgenommen sind anders beschriebene und die Sonst-Übergänge.
Die Statemaschine verwendet eine Heuristik, um unbekannte oder teilweise erkannte Namen zu vervollständigen. Die Heuristik ist nützlich, wenn Namensworte nicht in den Set-Datenstrukturen vorhanden sind. Die Heuristik fragt den ersten Buchstaben des nächsten Wortes ab. Ist dieser ein Großbuchstabe, wird das Wort dem aktuellen Teilnamen hinzugefügt, andernfalls wird so verfahren, wie es in Abbildung 2 dargestellt ist. Das Prüfen der Heuristik geschieht immer zuletzt pro Zustand und wird nur angewendet, wenn bereits ein Teilname erkannt wurde. Der Knoten ANY

kennzeichnet Worte, die durch diese Heuristik als Namensworte erkannt wurden. Zusätzlich wird die aktuelle Länge des gefundenen Teilnamens und auf Interpunktion getestet. Entspricht die Länge dem Längenlimit oder hat das zuletzt betrachtete Wort eine Interpunktion am Ende, werden am Teilnamen hinten anhängende Mittelnamen entfernt und der Name wird als gefunden getaggt. Das Standard-Namenslimit liegt bei -1, welches eine unendliche Länge kennzeichnet.
Die Statemaschine kann keine Namen der Form *Nachname, Vorname* erkennen. Weiterhin werden nur Präfix-titel betrachtet. Suffix-titel wie zum Beispiel *Dennis Armbruster, B.S.* werden nicht als Teil des Namen erkannt.

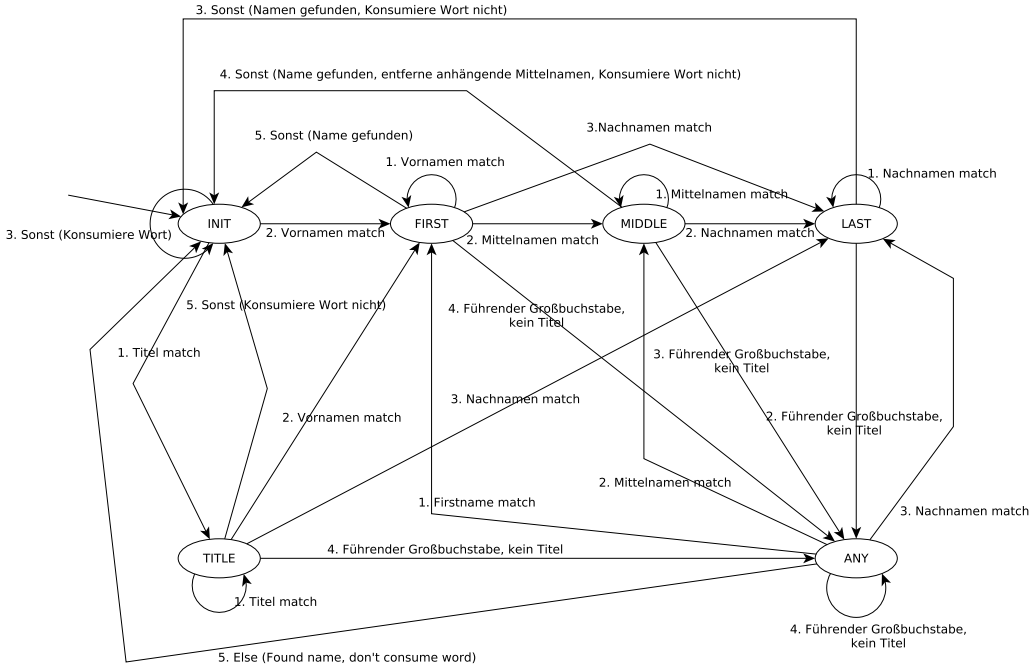

Abbildung 2: Der dargestellte Graph zeigt anschaulich die verwendete Statemaschine für die Namenserkennung. Die Zustände beschreiben das zuletzt gelesene Wort. Die meisten nicht Sonst-Transitionen konsumieren genau ein Wort, falls es nicht ausdrücklich anders beschrieben ist. Eine Transition kann nur gewählt werden, wenn ihre Bedingung erfüllt ist. Die Nummer der Transition gibt die Überprüfungsreihenfolge der Bedingungen an. INIT ist der Startzustand.

3.2.2.2 Scores für Namen

Für Namen, die im Text einer Webseite gefunden wurden, werden verschiedene Scores oder Bewertungsmaße berechnet. Anhand dieser Scores wird die Liste der Webseiten, die einen Namen enthalten, sortiert. Wenn das Bewertungsmaß die Relevanz einer Webseite für einen bestimmten Namen richtig widerspiegelt, kann so eine Liste der Webseiten für einen Namen mit absteigender Relevanz erstellt werden. Als relevant für einen Namen gilt eine Webseite, wenn sie primär von diesem handelt. Beispiele dafür können Lebenslaufseiten einer Person mit diesem Namen und Listen von Mitarbeiternamen sein.
Das Namensfinder-Teilsystem implementiert dafür Worthäufigkeits- oder auch tf.idf-Maße [30]. Diese Maße spiegeln die Relevanz eines Namens für eine Webseite durch entsprechendes Zählen der vorkommenden Namen wider. Bei wenigen Namensvorkommen sind diese Maße schlecht. Da die verwendete Namensfinder-Implementierung (3.2.2.1) keine Relativpronomen und keinen "Ich-Kontext" in einem Text erkennt, können sich viele solcher in Texten ebenfalls negativ auf das tf.idf-Maß auswirken.
Es werden folgende Worthäufigkeitsmaße verwendet:

- Das tf.idf Maß. Es berechnet sich durch $tf \cdot idf$. Es gilt $tf = \#Name$ und steht für Termfrequency oder Worthäufigkeit. Es ist die Anzahl an gefundenen Namen in einem Text. $idf = \log_2(N/df)$ steht für die inverse Termfrequency. N ist die Anzahl an Dokumenten und f die Zahl der Dokumente, in denen $Name$ enthalten ist. Der idf Teil sorgt für eine Skalierung auf Seiten, die den $Name$ enthalten. Durch den Logarithmus werden kleine Werte nicht zu stark gewichtet [31].

- Das bm25 Maß. Es berechnet sich durch $tf^* \cdot idf$. Der idf Teil berechnet sich dabei gleich wie beim tf.idf Maß. $tf^* = tf \cdot (k+1)/(k \cdot (1-b+b \cdot DL/AVDL)+tf)$, wobei DL für die Länge des Dokuments in Worten, $AVDL$ die durchschnittliche Länge über alle Dokumente in Worten, tf für die Termfrequency und k, b für

optimierbare Parameter steht. Die Standardwerte sind $k = 1,75$ und $b = 0,75$ [31].

Der bm25 Score verwendet eine modifizierte tf^*. Diese soll sich wie die normale tf verhalten, also $tf^* = 0 \leftrightarrow tf = 0$, wenn sich tf erhöht, soll sich auch tf^* erhöhen, wenn $tf \to \infty$ soll $tf^* \to limit$ gelten und tf soll mit Parameter b auf $DL/AVDL$ skaliert werden. Siehe dazu die Plausibilitätsargumente von [31, S.15-16]. Bm25 ist ein State-of-the-Art Worthäufigkeitsmaß [32].

3.2.2.3 Namen-URL Datei

Als Ergebnis wird eine Datei mit zeilenweisen normalisierten Namen (3.2.2) und einer sortierten Liste aller Webseiten in denen dieser Name gefunden wurde mit entsprechenden Scores zurückgegeben (3.2.2.3).
Das Zeilenformat dieser Datei ist

```
<normalized_name><tab><url><space>"("<tf.idf>,<bm25>")"(<space><url><space>
"("<tf.idf>,<bm25>")")*
```

Dabei bezeichnet `<normalized_name>` einen wie in Abschnitt 3.2.2 beschriebenen normalisierten Namen, `<url>` eine URL, `<tf.idf>` einen tf.idf und `<bm25>` einen bm25 Score wie in Abschnitt 3.2.2.2 beschrieben.
Dieses Format ist für Menschen leicht lesbar. Es können beliebig viele weitere Scores in die runden Klammern hinzugefügt werden, ohne einen Parser verändern zu müssen.

3.2.2.4 QLever Kompatible Dateien

Beim Verarbeiten der Texte können zusätzlich Dateien erstellt werden, die von QLever für eine Text Indexerstellung benötigt werden [28, 29]. Das System erstellt daher iterativ beim Verarbeiten der Crawlseiten words-, docs- und eine ntriple- Datei.

- Die words-Datei enthält alle Wörter eines Textes. Zusätzlich bindet es eine ID für den Satz, eine Information zum zugehörigen Namens-Tag und einen Score

an alle Worte. Der zugehörige Namens-Tag wird an alle Namensworte gebunden. Präfixe zählen dabei nicht dazu. Der Score ist fix und bei allen Worten auf 1.

- Die docs-Datei enthalten alle Sätze und die zugehörige Satz-ID.

- Die ntriple-Datei enthält Referenzen zur URL und den Sätzen, die im Text einer Seite mit einer URL vorhanden sind. In einer erweiterten QLever Version können diese Information dann direkt mit verknüpft werden [33]. Folgende n-tripple erstellt das System beim Verarbeiten der Seiten:

  <Alexander_Fleming> <appears-in> <http://source.url> .
  <QLever-internal-function/record-NUM> <source-url> <http://source.url> .

  Wobei "record-NUM" die ID des Satzes ist, der "<Alexander_Fleming>" enthält. Das System vermeidet es Dopplungen zu schreiben.

3.3 URLs von Bildungseinrichtungen

Dieses Projekt enthält auch ein Python Script, mit welchem sich Einstiegspunkte für den Webcrawler (3.1) in Form von Listen mit URLs von Bildungseinrichtungen erstellen lassen.
Dafür wird eine Query (Listing 3.4) an ein WikiData backend gestellt. Diese fragt die offizielle URL von Entitäten ab, die direkte oder beliebig viele Unterklassen Instanzen der Entität für Bildungseinrichtungen sind.
Listing 3.4 zeigt die verwendete Query dafür. Dabei sind `wdt:P31`, `wdt:P279`, `wdt:P856` und `rdfs:label` Relationen, welche in dieser Reihenfolge für *Instanz_von*, *Unterklasse_von*, *offizielle_Webseite* und *Namen* stehen. `wd:Q5341295` ist die Entität *Bildungsorganisation*. Zusätzlich wird durch die Filteranweisung auf englische Namen gefiltert.
Durch Ausführen der Query in Listing 3.4 wird eine Ergebnis Tabelle mit Spalten

```
SELECT DISTINCT ?id ?label ?url WHERE {
  ?id wdt:P31/wdt:P279* wd:Q5341295 .
  ?id wdt:P856 ?url .
  ?id rdfs:label ?label .
  FILTER(lang(?label) = 'en')
} LIMIT 100000
```

Listing 3.4: Dieses Listing zeigt eine Query die IDs, englische Namen und die offizielle URL von Bildungseinrichtungen abfragt.

für WikiData ID, deren Name und der offiziellen URL dieser zurückgegeben. Es ist anzumerken, dass eine Ergebniszeile nur erscheint, wenn eine Bildungseinrichtung in WikiData als Entität vorhanden ist, direkt oder über beliebig viele Unterklassen Relationen eine Instanz von *Bildungsorganisation* ist und einen englischen Namen besitzt.

4 Theoretische Analyse

In diesem Abschnitt wird die Komplexität der beiden Teilsysteme getrennt voneinander analysiert. Es werden approximierte asymptotische Maschinenoperationen und Hauptspeicherbelegungen betrachtet.

4.1 Analyse des Webcrawler

Der Webcrawler (3.1) lädt Webseiten aus dem Internet herunter, verarbeitet und speichert diese in komprimierte Dateien.

4.1.1 Laufzeit des Webcrawlers

Der Download von Webseiten über das Netzwerk und die IO-Festplatten Operationen sind teuer und werden nachfolgend als ähnlich teuer behandelt. Für n Webseiten müssen maximal $5 \cdot n$ Downloads (3.1.3) mit einer Seitengröße der größten Seite von m vorgenommen werden. Im schlechtesten Fall bestehen alle n Seiten die URL Prüfungen mit Download (3.1.2.2). Diese Checks bestehen aus einem Charset Lookup mit Vergleich zweier in der Länge begrenzten Zeichenketten und einem Doctype Check auf der Seite mit im schlechtesten Fall der Länge m. Für den ersten Check gilt aufgrund der konstanten Längen und konstanter lookup Zeit $\mathcal{O}(1+1) = \mathcal{O}(1)$. Der zweite Check hat im schlechtesten Fall einen Rechenaufwand von $\mathcal{O}(m)$. Die Operationskosten der beiden Prüfungen sind nur ein Bruchteil der IO- und Netzwerkkosten, können aber

mit konstanten Faktor skaliert gesehen werden.
Für die Größe der in n Dateien zu schreibenden Daten m' gilt $m' \leq m$, da der gefilterte sichtbare Text nicht größer als die Rohdaten sein kann.
Daraus ergibt sich im schlechtesten Fall eine Zeitkomplexität von $\mathcal{O}(5 \cdot n \cdot m \cdot 1 \cdot m) + \mathcal{O}(n \cdot m) = \mathcal{O}(n \cdot m \cdot (1 + m)) = \mathcal{O}(n \cdot m^2 + \underbrace{n \cdot m}_{\leq n \cdot m^2}) = \mathcal{O}(2 \cdot n \cdot m^2) = \mathcal{O}(n \cdot m^2)$ IO-/ Netzwerkoperationen.

Für das Prüfen (3.1.2.1) und Verarbeiten (3.1.4) der Seite vor dem Speichern werden approximierte Maschinenoperationen betrachtet.
Das Prüfen der Seite besteht aus einem Check, welcher Zeichenketten mit jeweils begrenzter Länge vergleicht, dem Überprüfen des netlocs, zwei membership Tests auf Set-Datenstrukturen und einem Dictionary lookup mit Prüfen der *robots.txt*.
Für die Vergleiche von zwei begrenzten oder konstanten Zeichenketten gilt $\mathcal{O}(k) = \mathcal{O}(1)$, wobei k die größere Konstante/Begrenzung der beiden Zeichenketten ist. Das netloc Feld darf für DNS Namen maximal 253 Zeichen lang sein (1.1.4). Daraus ergeben sich im schlechtesten Fall 253 Vergleiche mit $\mathcal{O}(253) = \mathcal{O}(1)$. Set-Datenstrukturen haben eine lookup Zeitkomplexität von $\mathcal{O}(1)$ [13]. Das Nachschlagen eines Dictionary Elements hat Komplexität $\mathcal{O}(1)$ [13], das Prüfen der *robots.txt* mit x Einträgen benötigt $\mathcal{O}(x)$ Operationen. Da meistens weniger als 10 Einträge oder keine *robots.txt* vorhanden sind, kann dies mit $\mathcal{O}(1)$ im Averagecase abschätzen werden.
Für die Prüfungen der Seiten ergibt sich so eine Zeitkomplexität von $\mathcal{O}(1+1+2 \cdot 1 \cdot 1) = \mathcal{O}(1)$.
Das Verarbeiten der Seite mit Größe m' besteht aus mehreren Vorgängen, welche im schlechtesten Fall die gesamte Seite lesen müssen. Daraus ergibt sich eine Zeitkomplexität von $\mathcal{O}(k \cdot m') = \mathcal{O}(m')$ mit konstantem k. Wenn m die größte Seite ist, gilt $m' \leq m$ und somit eine Komplexität von $\mathcal{O}(m)$.
Für Prüfen und Verarbeiten von n Webseiten wie beschrieben ergibt sich daraus eine Zeitkomplexität von $\mathcal{O}(n \cdot 1 + n \cdot m) = \mathcal{O}(n \cdot (1 + m)) = \mathcal{O}(n \cdot m)$.

Für Download, Schreiben, Verarbeiten und Prüfen der Crawlseiten ergibt sich eine gesamte Zeitkomplexität von $\mathcal{O}(n \cdot m^2 \cdot k + \underbrace{n \cdot m}_{\leq n \cdot m^2}) = \mathcal{O}(n \cdot m^2)$, wobei k ein konstanter Faktor ist, der die Mehrkosten von IO- und Netzwerk- gegenüber Maschinenoperationen beschreibt. Wenn weiterhin m mit der Größe der größten Seite konstant ist, ergibt sich eine Zeitkomplexität von $\mathcal{O}(n)$.
In dieser Betrachtung wurden für Set- und Dictionary-Datenstrukturen die Averagecases betrachtet. Siehe dazu auch die amortisierten worstcase Kosten in der Python Dokumentation [13]. Auch vernachlässigt wurden die hinzufüge Operationen mit konstanter Anzahl an Elementen an verschiedenen Stellen. Diese Operationen haben wieder im Averagecase einen Aufwand von $\mathcal{O}(k) = \mathcal{O}(1)$. Weiterhin vernachlässigt wurden die Komprimierungskosten des verwendeten bzip2 Algorithmus pro Datei.

4.1.2 Speichernutzung des Webcrawlers

Der Webcrawler (3.1) hält zur Ausführung eine Set-Datenstruktur mit bereits besuchten Seiten, eine Dictionarystruktur mit geparsten *robots.txt* Informationen, zwei Sets mit der aktuellen und nächsten Ebene der Breitensuche und mehrere konstante Datenstrukturen wie einzelne Integers, Strings, Sets und Listen im Hauptspeicher.
Für die konstanten Datenstrukturen ergibt sich eine Speicherkomplexität von $\mathcal{O}(1)$ über die gesamte Laufzeit.
Das Set mit den besuchten Seiten benötigt für n besuchte Webseiten $\mathcal{O}(n)$.
Im schlechtesten Fall liegt jede dieser n URLs auf einer eigenen Subdomain, es müssen also n *robots.txt* Dateien im Speicher gehalten werden. Da diese Textdateien wieder beliebig viele Regeln haben können, meistens aber weniger als 10 oder gar keine, ergibt sich im Averagecase eine Komplexität von $\mathcal{O}(10 \cdot n) = \mathcal{O}(n)$.
Die Anzahl an URLs im Set der aktuellen Ebene ist m. Wenn pro Seite in der aktuellen Ebene maximal k URLs gefunden und in das Set der nächsten Ebene kommen, ergibt das im schlechtesten Fall eine Komplexität von $\mathcal{O}(\underbrace{k + ... + k}_{m-mal}) = \mathcal{O}(m \cdot k) = \mathcal{O}(m)$

für ein konstantes k.
Da immer nur 2 Level der Breitensuche im Speicher gehalten werden, die Breite der Level aber mit n wächst, ergibt das $\mathcal{O}(k + 2 \cdot n \cdot m + (2 \cdot n)) = \mathcal{O}(n \cdot m + \underbrace{n}_{\leq n \cdot m}) = \mathcal{O}(2 \cdot n \cdot m) = \mathcal{O}(n \cdot m)$, wobei k die konstanten Datenstrukturen über die gesamte Laufzeit sind, n die Anzahl der besuchten Seiten und m die wachsende Breite der Level ist. Dies kann weiter mit dem größeren von m oder n abgeschätzt werden. Dadurch ergibt sich eine Quadratische Speicherkomplexität von $\mathcal{O}(max(m, n)^2)$.

4.2 Analyse des Namensfinder

Der Namensfinder (3.2) liest komprimierte Crawldateien aus einem Verzeichnis, findet Namen im Text, berechnet Scores und sortiert das Ergebnis anhand dieser Scores.

4.2.1 Laufzeit des Namensfinder

Es müssen n Crawldateien mit einer maximalen Textlänge der Größe m gelesen werden. Diese IO-Operationen haben also im schlechtesten Fall eine Komplexität von $\mathcal{O}(n \cdot m) = \mathcal{O}(n)$ mit konstantem maximalen m für alle Seiten.
Im Text werden Substitutionen von Zeichen vorgenommen. Zudem wird der Text nach Leerzeichen in Worte getrennt. Dafür muss k Mal über den Text der Länge m' iteriert werden, wobei k konstant ist. Wenn m der größte Text in allen Seiten ist, gilt $m' \leq m$ und somit $\mathcal{O}(n \cdot k \cdot m) = \mathcal{O}(n)$ mit n Seiten und konstantem m.
Die Namensfinder-Statemaschine geht über die Liste der Worte im Text und führt pro Wort maximal drei Set membership Tests und ein Match gegen einen regulären Ausdruck aus. Im schlechtesten Fall in einem nicht INIT Zustand kommt das gleiche Wort, ohne konsumiert zu werden, wieder in den INIT Zustand, in welchem ein Match gegen einen regulären Ausdruck und ein Set membership Test durchgeführt wird. Alle anderen Möglichkeiten in der Statemaschine kommen mit weniger Checks aus

und können daher mit dem schlechtesten Fall abgeschätzt werden. Daraus ergibt sich für m' Worte eine Zeitkomplexität von $\mathcal{O}(m' \cdot (4+2)) = \mathcal{O}(m')$ mit $\mathcal{O}(1)$ für die Set lookups und das Matchen eines regulären Ausdrucks. Sei m wieder die längste Wortliste, für die gilt $m' \leq m$, dann ergibt sich ein Aufwand von $\mathcal{O}(n \cdot m) = \mathcal{O}(n)$ mit konstantem m über n Seiten.

Für das Berechnen der Scores werden k Operationen pro Seite benötigt. Seien t die vom Namensfinder gefunden Namen. Im schlechtesten Fall wurde jeder Name mindestens einmal in allen n Seiten gefunden. Also ergibt sich eine Zeitkomplexität von $\mathcal{O}(t \cdot n \cdot k) = \mathcal{O}(t \cdot n)$ für konstantes k.

Wenn wieder t die gefundenen Namen sind und im schlechtesten Fall jeder der t Namen in jeder der n Seiten vorkommt, müssen pro Name aus t alle n Seiten anhand deren Score sortiert werden. Für die Sortierung wird eine Komplexität von $n \log n$ angenommen. Daraus ergibt sich für den Sortierungsschritt ein Aufwand von $\mathcal{O}(t \cdot n \log n)$.

Lesen der Seiten, Textoperationen, Namen finden, Scores berechnen und anschließendes Sortieren haben also eine gesamt Komplexität von

$\mathcal{O}(\underbrace{k \cdot n}_{=n} + n + n + t \cdot n + t \cdot n \log n) = \mathcal{O}(\underbrace{3 \cdot n}_{=n} + t \cdot n + t \cdot n \log n) = \mathcal{O}(\underbrace{n}_{\leq t \cdot n \log n} + \underbrace{t \cdot n}_{\leq t \cdot n \log n} + t \cdot n \log n) = \mathcal{O}(3 \cdot t \cdot n \log n) = \mathcal{O}(t \cdot n \log n)$ mit konstantem k, welches die Mehrkosten von IO- zu Maschinenoperationen beschriebt.

Bei dieser Betrachtung wurden wieder einzelne Set und Dictionary lookup- und Hinzufügeoperationen vernachlässigt. Diese haben aber im Averagecase eine konstante Zeitkomplexität.

5 Evaluation

In diesem Kapitel werden verschiedene Evaluationstechniken und Ergebnisse zu diesen dargestellt. Der Webcrawler (3.1) und der Namensfinder (3.2) werden getrennt voneinander betrachtet.

5.1 Evaluation des Webcrawlers

Für den Webcrawler ist es sinnvoll zu bewerten wie vollständig Webseiten von Wissenschaftlern und Personen an Bildungseinrichtungen gesammelt wurden. Wenn die Webseite einer Person fehlt, kann der Person nie die richtige Seite zugeordnet werden. Das Fehlen einer Webseite kann zudem Einfluss auf die Scores andere Seiten haben, da Teile davon die gesamte Dokumentenmenge betrachten (3.2.2.2).
Diese Vollständigkeitsevaluation wird nachfolgend als Abdeckung oder Coverage bezeichnet.

5.1.1 Abdeckung/Coverage

Eine komplette Abdeckung kann überprüft werden, wenn alle Webseiten bereits vor dem Crawlen bekannt sind. Mit einer solchen *target* Menge an Dokumenten kann die Coverage einer *real* Dokumentenmenge durch $coverage = \frac{|real|}{|target|} \cdot 100$ in Prozent ausgedrückt werden.

In der Praxis ist es bei großen Datensätzen sehr aufwendig und fehleranfällig vorher alle Ergebnisse per Hand bereitzustellen. Zum Beispiel enthält der Crawl der Albert-Ludwigs-Universität Freiburg vom 09.02.2019 beginnend bei `http://www.uni-freiburg.de/` mit einer Tiefe von 7 bereits 330.161 Webseiten.
Eine weniger aufwendige Möglichkeit ist die Abdeckung einer Stichprobe *sample* auf einer echten Dokumentenmenge *real* zu betrachten. Diese Stichprobenabdeckung berechnet sich in Prozent durch die Anzahl aller Stichprobenelemente die in *real* vorkommen, geteilt durch die Kardinalität der Stichprobenmenge $coverage_{sample} = \frac{|\{s|s \in sample \wedge s \in real\}|}{|sample|} \cdot 100$. Wenn alle Elemente der Stichprobe in *real* enthalten sind, ergibt sich ein Wert von 100 %. Jedes fehlende Element verringert entsprechend den $coverage_{sample}$ Wert um $\frac{1}{|sample|} \cdot 100$ Prozent, bis dieser bei 0 ist.
Wenn die Stichprobe möglichst breit und nicht zu klein gewählt ist, spiegelt eine solche Stichprobe die Gesamtmenge gut wider [34]. In der Statistik wird auch von einer repräsentativen Stichprobe gesprochen. Ein solches $coverage_{sample}$ Maß benötigt vergleichsweise weniger Aufwand und ist aussagekräftig über die Vollständigkeit einer Dokumentenmenge.

5.1.2 Crawler Evaluation Modul

Der Webcrawler (3.1) bietet die Möglichkeit mit dem Konsolenargument `evaluate` eine Liste von URLs auf einem Verzeichnis von Crawldateien zu evaluieren. Es gibt die Anzahl an URLs die in der Liste, nicht aber im Crawlverzeichnis sind, die Länge der verwendeten Liste und die Anzahl an Crawldateien zurück.
Die Liste der URLs wird als eine Textdatei angegeben. Diese enthält zeilenweise eine URL oder einen mit `#` beginnenden Kommentar. Dopplungen sind erlaubt und werden durch eine Set-Datenstruktur entfernt.
Für das Testen eines Listeneintrags wird eine Set-Datenstruktur für alle Dateinamen aus dem Crawlverzeichnis angelegt und ein membership Test durchgeführt. Die Namen und die URLs aus der Liste werden auf die gleiche Weise maskiert. Subdomains `www`,

`ww2` und `ww3`, hinten anhängende `"/"` im path und alle scheme werden ignoriert.
Um aus den Ergebnissen die von diesem Script zurückgegeben werden die $coverage_{Sample}$ wie in Abschnitt 5.1.1 beschrieben in Prozent zu berechnen, berechnet man $coverage_{Sample} = (1 - \frac{missed}{total\_list}) \cdot 100$.

5.1.3 Stichproben

Für die Statistiken wurden diverse Stichproben für verschiedene Bildungseinrichtungen verwendet. Es werden Crawls für die Albert-Ludwigs-Universität Freiburg, The University of Minnesota und die Universität Stuttgart verwendet.

Für die Albert-Ludwigs-Universität Freiburg gibt es nachfolgende URL Listen:

- `"link_list_uni_freiburg_random.txt"`: Diese Datei enthält 96 URLs von Instituten der Biologie, Romanistik, Geschichte, Mathematik und Theologie. Alle URLs wurden wie im Quelltext verlinkt und per Hand der Liste hinzugefügt. Manche dieser URLs leiten auf Synonyme Ressourcen weiter.

- `"link_list_uni_freiburg_info.txt"`: Diese Datei enthält alle Webseiten von Personen, die an der Albert-Ludwigs-Universität Freiburg im Bereich der Informatik arbeiten. Die Datei wurde im Dezember 2018 per Hand angelegt und enthält nach Entfernen von Dopplungen 360 URLs.

Für die Universität Stuttgart werden nachfolgende URL Stichproben verwendet:

- `"link_list_uni_stuttgart_random.txt"`: Diese Datei enthält 186 URLs von Instituten der Chemie, Mathematik, Physik, Geschichte, Germanistik, Philosophie, Wirtschaft, des Sports und Bau- und Ingenieurwesens. Alle URLs wurden wie im Quelltext verlinkt und per Hand der Liste hinzugefügt.

- `"link_list_uni_stuttgart_info.txt"`: Diese Datei enthält 55 URLs von Instituten der Informatik. Sie wurden per Hand, wie im Quelltext verlinkt, hinzugefügt.

Folgende URL Listendateien werden für The University of Minnesota genutzt:

- `"link_list_uni_twincities_random.txt"`: Diese Datei enthält 230 URLs von Instituten der Biologie, Chemie, Mathematik, Physik, Pflege, Anthropologie, Wirtschaft, Anglistik, Geschichte, Religions- und Bildungswissenschaften und des Designs. Alle URLs wurden wie im Quelltext verlinkt und per Hand der Liste hinzugefügt.

5.1.4 Ergebnisse

Die nachfolgenden Crawls wurden zu Testzwecken für diese Arbeit angefertigt. Manche von ihnen werden mit den in Abschnitt 5.1.3 beschriebenen Stichproben evaluiert. Die Vergleiche sind in Tabelle 1 aufgelistet.

- `"freiburg-d7.2019-02-09"` ist ein Crawl der Albert-Ludwigs-Universität Freiburg vom 09.02.2019 beginnend bei `http://www.uni-freiburg.de/` mit einer Crawltiefe von sieben und umfasst 330.161 Seiten.
  Der Zeitaufwand betrug für diesen Crawl ca. 32 Stunden. Es wurden 20 Python Threads verwendet, die sich auf verwendete Download Wiederholungsverzögerungen (3.1.3) auswirken. Die Speichernutzung lag im höchsten Level bei weniger als 2 GB. Es ergaben sich die folgenden URL Frontiers pro Level:
  `{0: 170, 1: 3537, 2: 30711, 3: 83948, 4: 196427, 5: 165184, 6: 390037}`
  Die Größe des Frontiers spiegelt nicht die Anzahl an gespeicherten, sondern nur verarbeiteten Seiten wider.

- Der Crawl `"stuttgart-d7.partial-d7.2019-02-26"` umfasst 1.993.700 Webseiten der Universität Stuttgart. Er wurde beginnend von `https://www.uni-`

`stuttgart.de/` mit einer Tiefe von sieben erstellt. Das siebte Level ist unvollständig und wurde per Hand abgebrochen, da der Crawl zu groß wurde.
Dieser Crawl lief ca. zwei Wochen, mit zwei Python Threads und hat zum Zeitpunkt des Abbruchs ca. 21 GB Speicher benötigt. Es ergaben sich die folgenden URL Frontiers pro Level:
`{0: 240, 1: 2597, 2: 24174, 3: 84379, 4: 271937, 5: 5226862}`
Das angefangene nächste Level hatte eine Größe von 8.1611.841 URLs.

- `"twin-cities-t10-d7.2019-02-28"` ist ein Crawl der University of Minnesota vom 28.02.2019 mit einer Tiefe von sieben. Er umfasst 515.634 Seiten. Es wurden 53h mit zehn Python Threads für die Erstellung benötigt. Nachfolgende URL frontiers ergaben sich pro Level:
`{0: 37, 1: 971, 2: 19553, 3: 81539, 4: 194810, 5: 294038, 6: 560797}`
Die Speichernutzung lag über den gesamten Zeitraum bei weniger als 3 GB.

Tabelle 1 zeigt, dass die $coverage_{sample}$ für ausgewählte Stichproben durchschnittlich bei $94,95$ % mit einem Median von $94,55$ % liegt.

Tabelle 1: Diese Tabelle zeigt die fehlenden Seiten und die $coverage_{Sample}$ Werte pro Stichprobe. Die Stichproben wurden gegen entsprechend Crawls der Bildungseinrichtungen verglichen.

sample	*missed*	$coverage_{Sample}$
`link_list_uni_freiburg_random.txt`	10	89,58 %
`link_list_uni_freiburg_info.txt`	2	99,44 %
`link_list_uni_stuttgart_random.txt`	14	92,47 %
`link_list_uni_stuttgart_info.txt`	3	94,55 %
`link_list_uni_twincities_random.txt`	3	98,70 %

Bei der Stichprobe `"link_list_uni_freiburg_random.txt"` enthalten 5 von 10 nicht gefundenen Seiten Dezimalpunkte im letzten Teil des path der URL ohne schließendes `"/"`. Solche URLs werden vom Crawler als Dateien mit Dateiendung für nicht HTML

Inhalte betrachtet und ignoriert.
Bei beiden fehlenden Seiten aus der Stichprobe `"link_list_uni_freiburg_random.txt"` ist die Tiefe von 7 nicht ausreichend um diese Seiten zu finden. Beide Webseiten sind in einem tieferen Crawl enthalten.
12 der 14 nicht vorhandenen Seiten aus `"link_list_uni_stuttgart_random.txt"` befinden sich auf der gleichen Subdomain. Der Crawl enthält keine Seiten dieser Subdomain. Das kann verschiedene Gründe haben, z. B. zeitlich begrenzte Unerreichbarkeit, eine *robots.txt* die den Zugang einschränkt oder eine zu geringe Tiefe.
Alle 3 fehlenden Mitarbeiterseiten aus der Stichprobe `"link_list_uni_twincities_random.txt"` befinden sich auf der gleichen Subdomain. Sie werden in der verlinkenden Webseite (`https://chem.umn.edu/people/staff`) in Listenseiten, die HTML Queries nutzen, dargestellt. Daher ignoriert der Crawler diese Webseiten.
Weitere Fehlerquellen, neben den bereits genannten, können relative URLs sein. Wenn im HTML einer Seite durch einen `<base>` Tag ein Präfix für relative URLs angegeben wird, der nicht gleich dem aktuellen Pfadpräfix ist, kann es zu fehlerhaften URL Zusammenfügungen kommen. Der Crawler liest `<base>` Tags vor dem Auflösen relativer URLs nicht aus.

6 Zukünftige Arbeiten

Das System kann an diesem Punkt wieder für die beiden Teilsystem getrennt betrachtet werden. Viele der hier aufgeführten Punkte hätten mit einem zeitlich größeren Rahmen bereits in dieses System mit eingebaut werden können.

6.1 Erweiterung des Webcrawlers

Der aktuelle Webcrawler verarbeitet nur HTML Inhalte. Für die Namensfindung könnten auch PDF oder ähnliche Formate interessant sein. Hierfür müsste ein solches System Text aus diesen Dateien lesen können.
Die content-type Überprüfung aus Abschnitt 3.1.2.2 kann durch eine Head-Request beschleunigt werden. Der Webserver muss diese unterstützen.
Es wird von den einzelnen Webseiten nur der sichtbare Text gespeichert. Durch dieses Verhalten gehen viele Informationen verloren. Zum Beispiel können Titel, Überschrift oder Listen HTML Anweisungen weitere Informationen zur Gewichtung von Namen in deren eingeschlossenem Text geben.
Beim Download über das Netzwerk kann eine echte Parallelisierung von Vorteil sein. Python unterstützt solch eine Parallelisierung mit ihrem multiprocessing Modul [35].
Es wird eine Set-Datenstruktur mit bereits besuchten URLs verwendet. Viele Seiten haben Synonyme URLs, die alle als unterschiedlich vom System betrachtet werden, aber auf den gleichen Inhalt verweisen. Das Speichern eines Hashwertes für die Seite oder eine andere Inhaltsprüfung könnte dieses Problem umgehen.

Der Crawler ignoriert query und Fragmente in URLs. Viele Webseiten verwenden diese, um auf einzelne Mitarbeiter zu verweisen. Durch ein Berücksichtigen solcher Seiten können mehr Mitarbeiterwebseiten gefunden und diesen Namen zugeordnet werden.
Der Crawler liest HTML Anweisungen, die eine Basis URL spezifizieren nicht. Dadurch kann zu Fehlern mit relativen Links kommen, wenn eine URL anders lautet, als in ihrem HTML als Basis URL angegeben.
Viele Webseiten benennen die Seiten ihrer Mitarbeiter mit einem Dezimalpunkt als Trenner zwischen den Namensteilen. Der Crawler versteht diese, wenn sie im letzten Teil des URL path ohne schließendes "/" vorkommen, als Dateityp und filtert sie auf eine Menge mit erlaubten Typen. Durch dieses Verhalten gehen Seiten verloren. Eine Blacklist für bestimmte Dateiendungen wie PDF oder Bildformate anstatt einer Whitelist zu verwenden könnte diese Fehler vermeiden.

6.2 Erweiterung des Name Finders

Das Finden von Namen in Text geschieht in dieser Implementierung des Systems mithilfe einer Statemaschine die extrahierte Namensteilen aus WikiData nutzt. Diese Namensteile müssen exakt so im Text vorkommen, um erkannt zu werden. Bessere Ergebnisse könnten mit einer toleranteren Maschinelearning Lösung erreicht werden. Bisher gruppiert das System Namen in einer Seite nicht. In einer erweiterten Version könnten lokal oder auch global über alle Seiten sehr ähnliche Namen zu einem zusammengefasst werden. Eine Möglichkeit lokal in einem Webseitentext zu gruppieren könnte das Zuordnen kürzerer Name zu längeren Namen anhand gleicher Namensworte sein. Solche Gruppierungen könnten sich positiv auf das Berechnen von Scores für eine Seite auswirken.
Die verwendete Statemaschine kann keine Suffix-titel und Namen der Form *Nachname, Vorname* erkennen. In vielen Webseitentexten kommen Namen in diesen Formen vor. Eine Erkennung solcher Namen kann sich positiv auf die richtige Zuordnung von

Namen-Webseitenpaaren auswirken.

Zudem ist das verwendete Relevanzmaß durch tf.idf Scores nicht optimal. Die reine Namenshäufigkeit kann zum Beispiel in Webseiten mit Namen in Aufzählungen wenig relevant sein. Auch Seiten mit sehr wenig Text und einem Namen sind unter Umständen nicht sehr relevant für eine Person.

Das Ausgabeformat in einer Textdatei ist für Menschen leicht lesbar. Ein strukturiertes Format wie N-Triple zu verwenden ist sinnvoll, wenn aus den Ergebnissen ein Index gebaut werden soll.

Eine Evaluation des Namensfinder in Gesichtspunkten der Genauigkeit und Präzision sollte in einer Erweiterung dieser Arbeit in Betracht gezogen werden.

7 Zusammenfassung

Das Crawlen von Mitarbeiterwebseiten kann durch nicht eindeutige Dateinamen und -formate, tiefe Seitenstrukturen, unterschiedliche Designs und Erreichbarkeitsschwankungen durch das Internet schwierig sein. Das in dieser Arbeit vorgestellte System löst diese Probleme durch verschiedene Ansätze und findet in entsprechenden Crawls durchschnittlich $94,95$ % aller Mitarbeiterseiten mit einer angemessenen Ressourcennutzung (5.1).

Für Namen, die in den Crawlseiten gefunden wurden, erstellt das System eine in Relevanz absteigende Liste an Webseiten für einen Namen. Es werden dafür State-of-the-Art Worthäufigkeitsmaße verwendet.

Wie im Abschnitt 6 Zukünftige Arbeiten dargestellt, gibt es noch einige Punkte die in beiden Teilsystemen verbessert und erweitert werden könne.

Literaturverzeichnis

[1] A.-L.-U. Freiburg, "Zahlen uni freiburg." https://www.uni-freiburg.de/universitaet/portrait/universitaet-in-zahlen, 2019 (accessed Februar 28, 2019).

[2] W. Foundation, "Webcrawler." https://de.wikipedia.org/wiki/Webcrawler, 2019 (accessed Februar 22, 2019).

[3] robotstxt.org, "Robots exclusion protocol." http://www.robotstxt.org/robotstxt.html, 2019 (accessed Februar 22, 2019).

[4] W. Foundation, "Wikidata." https://www.wikidata.org/wiki/Wikidata:Main_Page, 2019 (accessed Februar 20, 2019).

[5] W. Community, "Resource description framework." https://de.wikipedia.org/wiki/Resource_Description_Framework, 2018 (accessed August 7, 2018).

[6] T. Berners-Lee, L. Masinter, X. Corporation, M. McCahill, and U. of Minnesota, "Uniform ressource locators." https://tools.ietf.org/html/rfc1738, 2019 (accessed Februar 22, 2019).

[7] U. I. R. Fielding, "Relative uniform ressource locators." https://tools.ietf.org/html/rfc1808.html, 2019 (accessed Februar 22, 2019).

[8] U. I. R. Fielding, "Relative uniform ressource locators." https://en.wikipedia.org/wiki/Domain_Name_System#Domain_name_syntax,_internationalization, 2019 (accessed Februar 22, 2019).

[9] O. Heinonen, K. Hätönen, and M. Klemettinen, "Www robots and search engines," 1996.

[10] S. Chakrabarti, M. van den Berg, and B. Dom, "Focused crawling: a new approach to topic-specific web resource discovery," *Computer Networks*, vol. 31, no. 11, pp. 1623 – 1640, 1999.

[11] G. Pant and P. Srinivasan, "Learning to crawl: Comparing classification schemes," *ACM Trans. Inf. Syst.*, vol. 23, pp. 430–462, Oct. 2005.

[12] IANA, "Internet assigned numbers authority." https://www.iana.org/, 2019 (accessed Februar 22, 2019).

[13] J. Hartley, "Time complexity python builtins." https://wiki.python.org/moin/TimeComplexity, 2019 (accessed Februar 19, 2019).

[14] B. Kleineidam, "Cpython robotparser.py." https://github.com/python/cpython/blob/3.7/Lib/urllib/robotparser.py, 2019 (accessed Februar 22, 2019).

[15] Wikipedia, "Time complexity python builtins." https://de.wikipedia.org/wiki/UTF-8, 2019 (accessed Februar 19, 2019).

[16] Wikipedia, "Wikipedia aufbau des http user agent." https://de.wikipedia.org/wiki/User_Agent#Aufbau_des_HTTP-User-Agent-String, 2019 (accessed Februar 19, 2019).

[17] Mozilla, "Mozilla user agent." https://developer.mozilla.org/de/docs/Web/HTTP/Headers/User-Agent, 2019 (accessed Februar 19, 2019).

[18] Wikipedia, "Wikipedia identifikation durch den user agent." https://de.wikipedia.org/wiki/Mozilla#Teil_der_Identifikation_vieler_Webbrowser, 2019 (accessed Februar 19, 2019).

[19] Wikipedia, “Wikipedia http-statuscode.” https://de.wikipedia.org/wiki/HTTP-Statuscode, 2019 (accessed Februar 19, 2019).

[20] certifi, “certifi.” https://pypi.org/project/certifi/, 2019 (accessed Februar 19, 2019).

[21] L. Richardson, “Beautifulsoup.” https://www.crummy.com/software/BeautifulSoup/bs4/doc/, 2019 (accessed Februar 20, 2019).

[22] R. Fieldinga and U. Irvine, “Fragments in url.” https://tools.ietf.org/html/rfc1808.html#section-2.1, 2019 (accessed Februar 20, 2019).

[23] P. S. Foundation, “bz2.” https://docs.python.org/3/library/bz2.html, 2019 (accessed Februar 20, 2019).

[24] L. Collin, “Compression benchmark 0.” https://tukaani.org/lzma/benchmarks.html, 2019 (accessed Februar 20, 2019).

[25] A. one x86, “Compression benchmark 1.” https://catchchallenger.first-world.info/wiki/Quick_Benchmark:_Gzip_vs_Bzip2_vs_LZMA_vs_XZ_vs_LZ4_vs_LZO, 2019 (accessed Februar 20, 2019).

[26] W. Foundation, “Filename reserverd characters.” https://en.wikipedia.org/wiki/Filename#Reserved_characters_and_words, 2019 (accessed Februar 20, 2019).

[27] R. Fielding and U. Irvine, “Bnf url.” https://tools.ietf.org/html/rfc1808.html#section-2.2, 2019 (accessed Februar 20, 2019).

[28] ad freiburg, “Qlever.” https://github.com/ad-freiburg/QLever/blob/master/README.md, 2019 (accessed Februar 20, 2019).

[29] ad freiburg, “Qlever + text.” https://github.com/ad-freiburg/QLever/blob/master/docs/sparql_plus_text.md, 2019 (accessed Februar 20, 2019).

[30] W. Foundation, "Worthäufigkeitsmaße." https://de.wikipedia.org/wiki/Tf-idf-Ma%C3%9F, 2019 (accessed Februar 22, 2019).

[31] P. D. H. Bast, "Informationretrieval ws1718 lecture-02." https://daphne.informatik.uni-freiburg.de/ws1718/InformationRetrieval/svn/public/slides/lecture-02.pdf, 2019 (accessed Februar 22, 2019).

[32] W. Foundation, "Wikipedia bm25." https://en.wikipedia.org/wiki/Okapi_BM25, 2019 (accessed März 01, 2019).

[33] N. Schnelle, "Qlever proposal association of text records and kb data." https://github.com/ad-freiburg/QLever/issues/175, 2019 (accessed Februar 22, 2019).

[34] W. Foundation, "Wikipedia hochrechnung." https://de.wikipedia.org/wiki/Hochrechnung, 2019 (accessed Februar 26, 2019).

[35] P. S. Foundation., "Python multiprocessing." https://docs.python.org/3.7/library/multiprocessing.html?highlight=process, 2019 (accessed Februar 25, 2019).