Analyse und Darstellung von Hypertextstrukturen mithilfe eines in Python programmierten Webcrawlers

Eine W-Seminararbeit

Faris Avdic
[redacted]

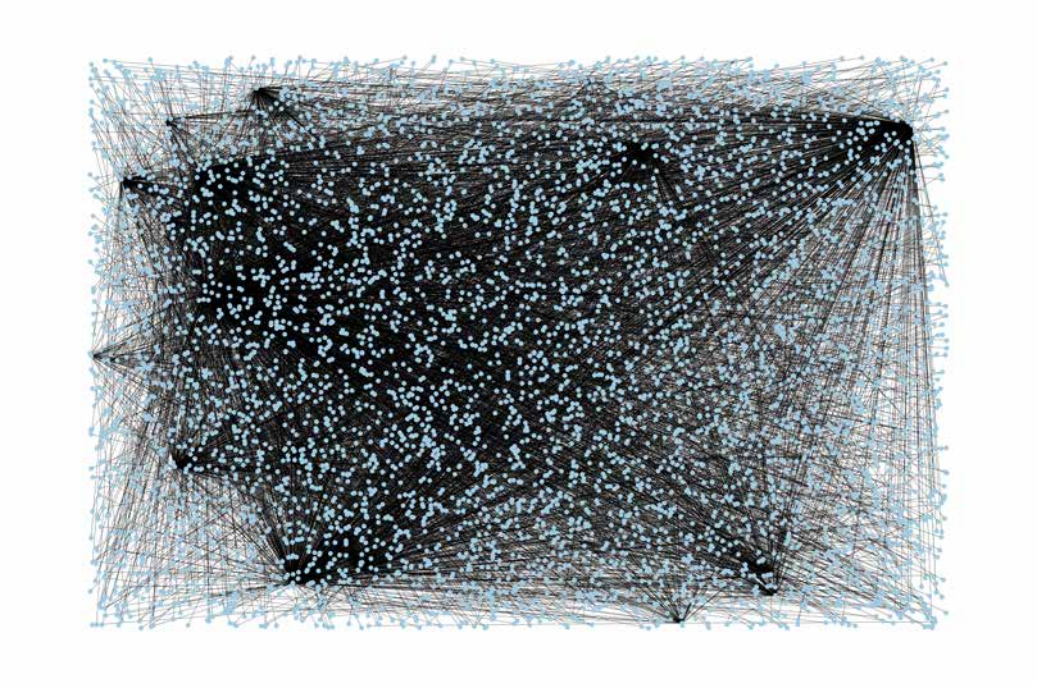

Abbildung 1: Visualisierung eines Graphen aus ca. 16.000 Knoten, eigene Darstellung

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

> Welcome to the internet
> Have a look around
> Anything that brain of yours can think of can be found
> We've got mountains of content
> Some better, some worse
> If none of it's of interest to you, you'd be the first

Mit diesen Versen beginnt Bo Burnham das Lied „Welcome to the Internet“ [Bur21], ein satirischer Text, der sich mit der Reizüberflutung, die die Menschheit durch das Internet erfährt, auseinandersetzt [gen21].
Das Internet bietet den Menschen Zugang zu einer geradezu unendlichen Menge an Informationen, zu jedem Thema, zu jeder Zeit und an jedem Ort. Dieser ständige Zugang zu Informationen ist allerdings nicht selbstverständlich, er ist das Ergebnis von Suchmaschinen und ihren Webcrawlern, die ununterbrochen das gesamte Internet nach neuen Webseiten durchsuchen und sie indexieren [YMH02, S. 1].
Ein solcher Webcrawler soll im Rahmen dieser Arbeit in der Programmiersprache Python entwickelt werden, um die Online-Enzyklopädie Wikipedia zu indexieren. Mithilfe der dabei gesammelten Daten wird Wikipedia anschließend auf strukturelle Besonderheiten untersucht. Neben dem bloßen Aufbau Wikipedias sollen auch Untersuchungen bezüglich der Möglichkeit des Auffindens neuer inhaltlicher Zusammenhänge zweier, bei erster Betrachtung beziehungsloser, Ereignisse angestellt werden. Daneben sollen die Struktur Wikipedias auch anschaulich visualisiert werden (s. Abbildung 1).

2 Begriffserklärungen

Im Folgenden werden einige zum Verständnis der Arbeit wichtige Begriffe erklärt, sowie polyseme Begriffe definiert und erläutert.

2.1 Webcrawler

Webcrawler sind Programme, die Webseiten nach Hyperlinks durchsuchen und rekursiv die so gefundenen Webseiten untersuchen [Naj09, S. 1]. Dabei sind neben den Links oft auch der rohe Text, E-Mail Adressen oder Telefonnummern von Interesse.

2.2 Graphentheorie

Die Graphentheorie ist ein Teilbereich der Mathematik und Informatik und beschäftigt sich mit Graphen und ihren Beziehungen zueinander.

Datenstruktur Graph Der Graph als Datenstruktur unterscheidet sich stark von Funktionsgraphen, wie sie den meisten aus der Mathematik bekannt sind. In der Graphentheorie ist ein Graph G das Paar zweier Mengen $G = (V, E)$.
Dabei beschreibt $V = \{v_0, v_1, v_2, ..., v_{n-1}\}$ die Menge aller n Knoten des Graphen.
$E = \{e_0, e_1, e_2, ..., e_{m-1}\}$ mit $e_i = (v_a, v_b)$ beschreibt die Menge aller m Kanten, die die Knoten des Graphen miteinander verbinden. So verbindet e_i die Knoten v_a und v_b miteinander [Die06, S. 2].
Abbildung 2 ist die graphische Darstellung eines einfachen Graphen.

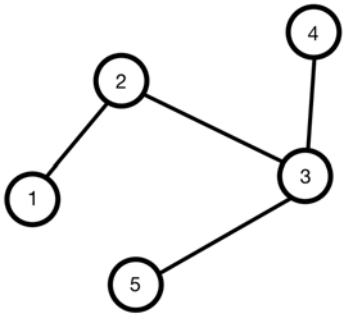

Abbildung 2: Diagramm eines einfachen Graphen, eigene Darstellung

Wenn im Folgenden von Graphen gesprochen wird, so sind, sofern nicht explizit anders festgelegt, Graphen als Datenstruktur gemeint.

Eigenschaften von Graphen Graphen lassen sich durch diverse Eigenschaften charakterisieren. So wird ein Graph als „gewichtet“ bezeichnet, wenn jeder Kante des Graphen eine reelle Zahl als Kantengewicht zugeordnet wird [KN09, S. 74 f.]. In dieser Arbeit wird nur mit ungewichteten Graphen gearbeitet, das heißt, dass jede Kante das Kantengewicht 1 hat.
Neben dem Kantengewicht kann man Graphen auch danach charakterisieren, ob ihre Kanten gerichtet oder ungerichtet sind. Ist ein Graph gerichtet, so hat jede Kante des Graphen eine feste Richtung. Befindet sich in einem gerichteten Graphen die Kante $e_i = (v_a, v_b)$, so verbindet sie nur den Knoten v_a mit v_b, nicht aber v_b mit v_a. In einem ungerichteten Graphen verläuft eine

Kante immer in beide Richtungen [KN09, S. 7 f.].
Es gibt eine Vielzahl weiterer Charakterisierungsmöglichkeiten für Graphen, die für diese Arbeit aber nicht von Relevanz sind und deswegen nicht weiter erläutert werden.

Zyklus Als Zyklus (Pl. Zyklen) wird in der Graphentheorie ein Pfad bezeichnet, der über verschiedene Kanten wieder zum Anfangsknoten führt. Zyklen, die aus nur zwei Knoten bestehen, werden als trivial bezeichnet und bei der Untersuchung eines Graphen meist nicht beachtet [Die06, S. 7 ff.]. Ein trivialer Zyklus ist in Abbildung 3 zu sehen, Abbildung 4 zeigt einen nicht-trivialen Zyklus aus 3 Knoten (v_2,v_3 und v_4).

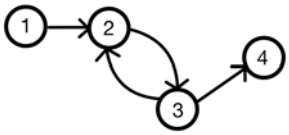

Abbildung 3: Graph mit trivialem Zyklus, eigene Darstellung

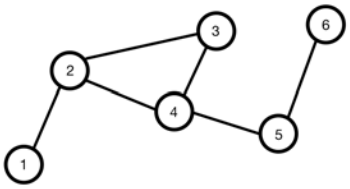

Abbildung 4: Graph mit nicht-trivialem Zyklus, eigene Darstellung

Knotengrad Der Knotengrad beschreibt die Anzahl der an einen Knoten angrenzenden Kanten [Die06, S. 5 f.]. So hat der Knoten v_3 in Abbildung 3 den Grad 3 und der Knoten v_5 in Abbildung 4 den Grad 2.

3 Entwicklung eines Webcrawlers in Python

3.1 Funktionsweise und Anwendungen von Webcrawlern

Webcrawler sind aus dem Leben im 21. Jahrhundert nicht mehr wegzudenken. Suchmaschinen, wie Google und bing, nutzen Webcrawler um das Internet zu indexieren und uns so einen komfortablen Zugriff auf Informationen und Dienste aller Art bieten. Ohne diese Suchmaschinen, wäre es unmöglich nach einem bestimmten Begriff im Internet zu suchen [Tur10, S. 10 f.]. Stattdessen müsste man aus einem Verzeichnis von URLs einige Webseiten heraussuchen, die eventuell die gesuchte Information irgendwo gespeichert haben und dann manuell jede dieser Webseiten nach der gesuchten Information absuchen. Eine überaus langwierige und auch anstrengende Methode, die der Menschheit

zum Glück erspart bleibt. Durch die Arbeit, die Google und Co. jeden Tag leisten, ist es möglich, dass man innerhalb von Sekundenbruchteilen mehrere Tausend Webseiten vorgeschlagen bekommt, die eine gesuchte Information mit Sicherheit in irgendeiner Form bieten können. Selbstverständlich geht die Arbeit der großen Suchmaschinen weit über die bloße Indexierung des Internets hinaus, mithilfe von künstlicher Intelligenz sind mittlerweile Dinge möglich, von denen man in den Zeiten der ersten Webcrawler nicht einmal träumen konnte.

Trotz der enorm großen Rolle, die Webcrawler in unserem Leben spielen, ist die grundlegende Funktionsweise dieser erstaunlich simpel. Die allgemeine Architektur von Webcrawlern besteht nur aus wenigen Teilen und ist auch in Abbildung 5 als Schaubild dargestellt.

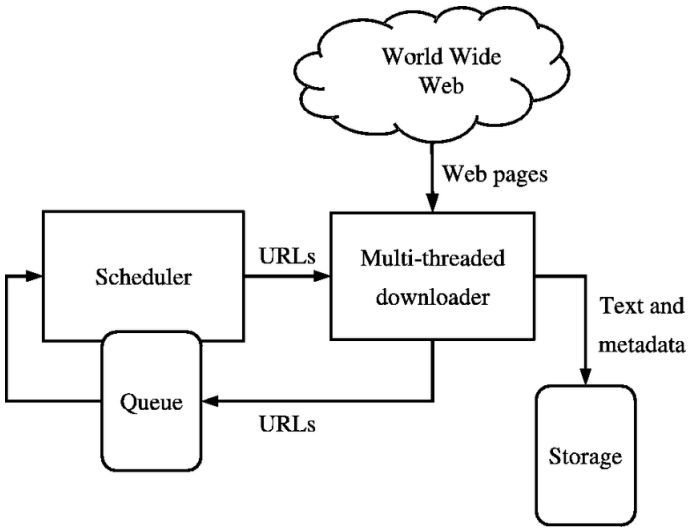

Abbildung 5: Allgemeiner Aufbau von Webcrawlern [Cas04, S. 35]

Der Webcrawler startet mit einer Liste von wichtigen URLs, von denen die Indexierung ausgehen soll. Diese Liste bezeichnet man als Seed. Dieser Seed wird als nächstes den „Multi-Threaded-Downloader“ übergeben, ein Programm, dass die Webseiten aus dem Internet herunterlädt. Aus dem HTML-Code werden dann URLs und gewünschte Informationen extrahiert. Die URLs landen in einer Warteschlange, die anderen Daten werden gespeichert und weiterverarbeitet. Der „Scheduler“ priorisiert die URLs und gibt sie umgeordnet wieder an „Multi-Threaded-Downloader“ weiter, woraufhin sich der gesamte Prozess wiederholt. Die Indexierung wird unterbrochen, wenn eine Abbruchbedingung eintritt. Diese kann das Erreichen einer gewünschten

Anzahl von Webseiten oder die Auslastung der zur Verfügung stehenden Kapazitäten sein.[Cas04, S. 34 f.]

Dies ist nur eine generelle Architektur für Webcrawler, je nach individuellen Ansprüchen können diese auch deutlich komplexer oder einfacher ausfallen.

3.2 Architektur des entwickelten Webcrawlers

Wie im vorherigen Kapitel bereits erklärt, kann die Komplexität und der Aufbau von Webcrawlern in Abhängigkeit der individuellen Ansprüche stark variieren. Der Webcrawler, der im Rahmen dieser Arbeit entwickelt wurde setzt einige Elemente dieser Struktur nur primitiv um, was auf einige Umstände zurückzuführen ist, die eine weniger komplexe Implementierung ermöglichen.

Eine erste Vereinfachung liegt darin, dass der Webcrawler bereits besuchte Webseiten nicht erneut ansteuern muss, um sie zu aktualisieren. Der Graph, den der Webcrawler erstellen soll, hat keinen Anspruch an Aktualität, sodass eine „Momentaufnahme“ des Netzwerks vollkommen ausreicht. Diese kann auch bei einmaligem Besuch der Webseiten erstellt werden.
Dazu kommt der vergleichsweise winzige Bereich des Internets, in dem der Webcrawler arbeitet. Denn anders als die Webcrawler großer Suchmaschinen, beschränkt sich das Ressort dieses Webcrawlers auf den deutschsprachigen Teil Wikipedias, der „nur“ aus etwa 2,6 Millionen Webseiten besteht [Wik21a].
In diesem eingeschränkten Arbeitsbereich liegt auch die dritte Vereinfachung. Der Webcrawler ist so programmiert, dass er nur die Links findet, die zu anderen Wikipedia-Artikeln führen, Links von anderen Webseiten oder speziellen Wikipedia-Seiten, wie Diskussionsforen oder Bildern, führen, werden nicht beachtet. Diese Wikipedia-Artikel haben den Vorteil, dass sie ohne jegliche Hierarchie angeordnet sind, sie sind also alle gleichwertig. Das eliminiert die Notwendigkeit einer Priorisierung der URLs durch den Webcrawler, der Scheduler wird entlastet.

Diese Umstände ermöglichen eine Vereinfachung der Webcrawler-Architektur. So besteht der Seed dieses Webcrawlers immer aus nur einer URL, meistens der eines zufälligen Wikipedia-Eintrags. Durch den fehlenden Bedarf einer Priorisierung der gefundenen URLs kann der Scheduler auch gänzlich weggelassen werden. Stattdessen arbeitet der Webcrawler rekursiv alle gefundenen URLs ab. Der „Mutli-Threaded-Downloader“ ist stark vereinfacht durch eine einzige Funktion vertreten, die den gesamten HTML-Code einer Webseite herunterlädt. Anders als die Downloader kommerzieller Webcrawler hat die-

se Funktion keine Möglichkeit zur Parallelisierung, eine Programmiertechnik deren Komplexität diese Arbeit übersteigen würde (s. 5.1, „Optimierung“). Das Extrahieren der URLs ist bei diesem Webcrawler der komplexeste Teil, obwohl er ebenfalls nur durch eine einzige Funktion implementiert ist. Die Funktion nimmt den rohen HTML-Code einer Webseite und sucht aus diesem alle Hyperlinks heraus. Aus dieser Liste mit allen enthalten Hyperlinks werden die unerwünschten Links herausgefiltert, das heißt es bleiben nur noch die Links übrig, die zu anderen Wikipedia-Artikeln führen. Links, die zu Wikipedia-Diskussionsforen, Bildern oder anderen Webseiten führen, werden ignoriert. Die übrig geblieben Links werden zur Warteschlange hinzugefügt und der Prozess beginnt von vorne. Die Abbruchbedingung ist dann gegeben, wenn eine vorher bestimmte Anzahl an Rekursionsebenen erreicht ist. Der Prozess der Indexierung ist in Abbildung 6 noch einmal anschaulich dargestellt.

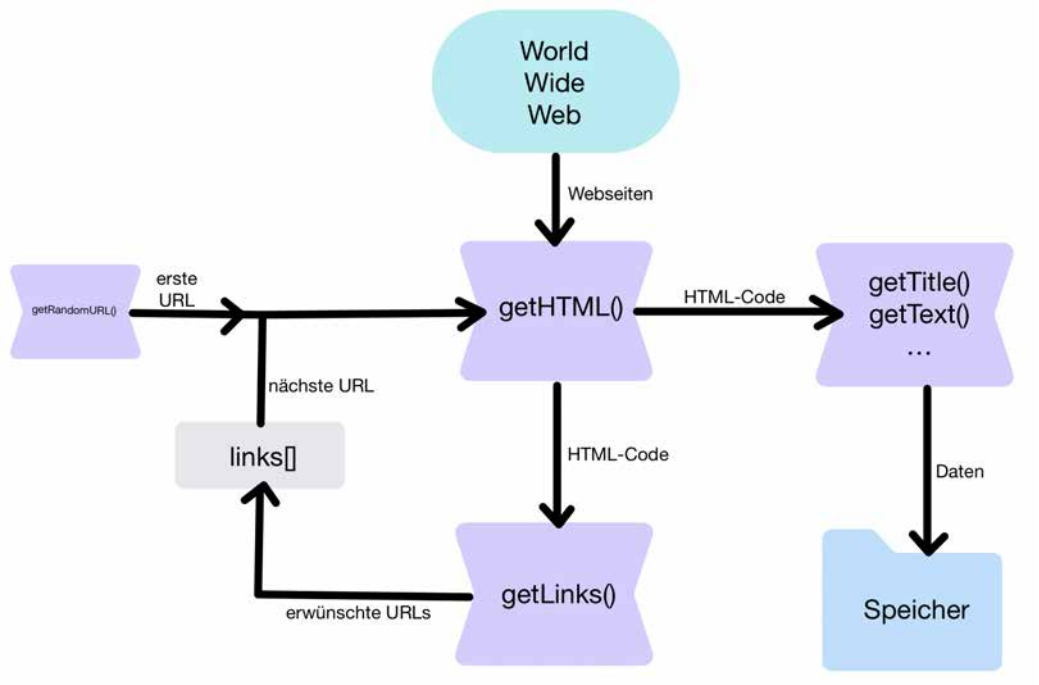

Abbildung 6: Diagramm des Indexierungsprozesses, eigene Darstellung nach [Cas04, S. 35]

Neben der Indexierung hat der Webcrawler die Aufgabe, die URLs der gefundenen Artikel in einem Graphen zu speichern. Diese Aufgabe wird durch einen einzigen Funktionsaufruf bei der Bearbeitung jedes einzelnen Artikels übernommen. Details zur Implementierung werden in Kapitel 3.3 erläutert.

3.3 Implementierung des Webcrawlers in Python

Das gesamte Projekt wurde objektorientiert programmiert. Grundstein der gesamten Indexierung ist die Klasse `Eintrag`, von der jedes Objekt jeweils einen anderen Wikipedia-Artikel repräsentiert. Attribute und Methoden der Klasse können dem Klassendiagramm in Abbildung 7 entnommen werden. Die Klasse `Webcrawler` führt die Indexierung und die weiteren Untersuchungen aus, die entsprechenden Attribute und Methoden sind im Klassendiagramm in Abbildung 8 aufgeführt.

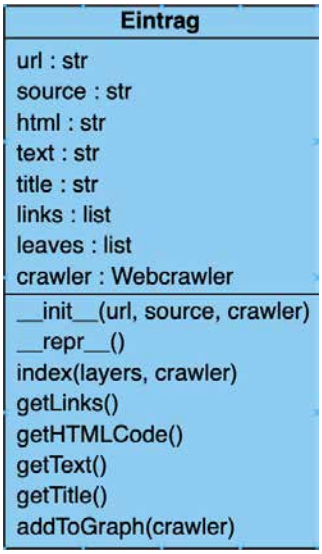

Abbildung 7: Klassendiagramm der Klasse Eintrag, eigene Darstellung

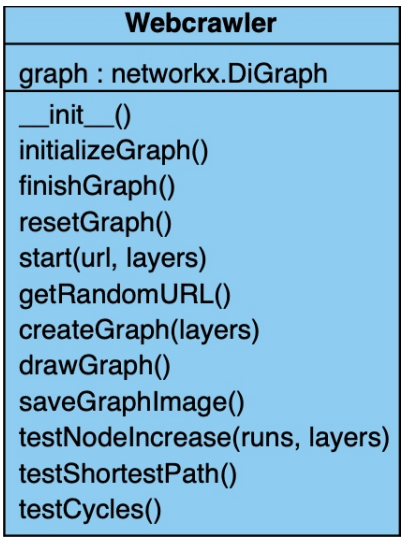

Abbildung 8: Klassendiagramm der Klasse Webcrawler, eigene Darstellung

Bibliotheken Der vergleichsweise einfache Code des Webcrawlers wird durch die Verwendung von zahlreichen Bibliotheken ermöglicht, die viele in der Realität überaus komplizierte Prozesse vereinfachen. Es folgt ein Überblick der verwendeten Bibliotheken mit einer kurzen Beschreibung des jeweiligen Einsatzbereichs.

`requests`
Das Modul `requests` wird verwendet um den HTML-Code einer Website herunterzuladen, sodass im Anschluss mit diesem gearbeitet werden kann.

`networkX`
Das Modul `networkX` stellt neben Klassen für viele Arten von Graphen auch einige wichtige Algorithmen aus der Graphentheorie zur Verfügung. Der Webcrawler verwendet die Klasse `DiGraph` der Bibliothek, um die gefundenen Einträge in einem gerichteten Graphen zu speichern. Außerdem werden die vordefinierten Algorithmen zur Untersuchung des Graphen verwendet.

`BeautifulSoup4`
Das Modul `BeautifulSoup4` (kurz: `bs4`) bietet einige Funktionen zum Parsen von HTML-Code. Obwohl die Bibliothek sehr mächtig ist, beschränkt sich ihre Anwendung hier auf die Suche nach den `<a>`-Tags im HTML-Code, die Hyperlinks repräsentieren.

`matplotlib`
Das Modul `matplotlib` ist der gängige Standard zur Visualisierung von Daten mit Python. In diesem Fall wird das Untermodul `pyplot` verwendet, um den Graphen zu zeichnen.

`os`
Das Modul `os` erlaubt einem Python-Programm mit dem Betriebssystem zu interagieren. Obwohl viel mehr möglich wäre, beschränkt sich der Einsatz des Moduls im Webcrawler auf die Überprüfung der Existenz, sowie das Löschen einer Datei.

`random`
Das Modul `random` ermöglicht die Ausgabe von zufälligen Werten. Hier wird der Befehl `sample` verwendet, um zufällige Knoten aus dem Graphen zu erhalten.

Die Arbeit des Webcrawlers lässt sich grob in drei Aufgabenbereiche unterteilen: die Beschaffung von Daten, die Verarbeitung dieser Daten und die Speicherung der Daten im Graphen.

Datenbeschaffung Bei der ersten Aufgabe ist die Arbeitsweise des Webcrawlers stark Kapitel 12 von Al Sweigarts Buch „Automate the Boring Stuff With Python“ [Swe19] nachempfunden. Die `get`-Funktion des `requests`-Moduls wird verwendet, um die Arbeit des Downloaders zu übernehmen und den gesamten HTML-Code hinter einer URL herunterzuladen. Dieser HTML-Code wird als String im Attribut `html` der Klasse `Eintrag` gespeichert. Damit

ist der Schritt der Datenbeschaffung erledigt, aus dem HTML-Code werden in den nächsten Schritten alle benötigten Informationen extrahiert. Abbildung 9 zeigt den simplen Code, der alle weiteren Untersuchungen erst möglich macht.

```
535     def getHTMLCode(self):
536         """
537         Gibt den HTML-Code des aktuellen Eintrags als String zurück.
538 
539         Rückgabe
540         --------
541         str
542             vollständiger HTML-Code des Eintrags
543         """
544         res = requests.get(self.url)
545         return res.text
```

Abbildung 9: Code der Funktion getHTMLCode(), eigene Darstellung

Datenverarbeitung Der nächste Schritt wird durch die Bibliothek `bs4` enorm erleichtert. Diese bringt einige Funktionen mit sich, die aus HTML-Code bestimmte Informationen herausfiltert. So gibt einem die Funktion `get_text()` den Verfassertext einer Seite zurück, ohne manuell die HTML-Tags oder andere unerwünschte Zeichen entfernen zu müssen. Der Rückgabewert von `get_text()` wird im Attribut `text` gespeichert. Die meisten anderen Attribute der Klasse `Eintrag` werden analog initialisiert (s. Abbildung 7), eine Ausnahme bietet das `links`-Attribut. Die Funktion, um dieses zu initialisieren ist um einiges komplexer aufgebaut. `bs4` bringt zwar die Funktion `findAll()` mit sich, mit der man nach jedem beliebigen HTML-Tag suchen kann, doch mit der Suche nach den <a>-Tags ist die Aufgabe noch nicht erledigt. Deswegen übernimmt die Funktion `getLinks()` der Klasse `Eintrag` auch die Rolle des URL-Extractors. Um die Regeln für diesen aufstellen zu können, wurden während der Entwicklung des Webcrawlers alle gefundenen Links in einer Textdatei gespeichert, was die Untersuchung unerwünschter URLs nach gemeinsamen Merkmalen erleichtert. So wurden immer mehr Links ausgeschlossen, bis nur noch erwünschte Links in den Textdateien zu finden waren. Das Speichern der Links in einer Textdatei wurde anschließend zur Laufzeitoptimierung wieder entfernt. Somit liefert einem die Funktion `getLinks()` der Klasse `Eintrag` eine Liste von allen erwünschten Links, die in einem Wikipedia-Artikel gefunden wurden als Rückgabewert. Diese Liste wird im Attribut `links` gespeichert. Abbildung 10 zeigt die Implementierung des beschriebenen Vorgangs.

```
452     def getLinks(self):
453         soup = BeautifulSoup(self.html, "html.parser")
454         allLinks = soup.findAll('a', href=True)
455 
456         #alle URLs (href-Attribute der <a>-Tags) werden in Liste hrefs
457         #gespeichert
458         hrefs = []
459         for l in allLinks:
460             hrefs.append(l["href"])
461 
462         links = []
463         for l in hrefs:
464             #filtert Links heraus, die zu anderen Internetseiten führen
465             #(nicht Wikipedia, z.B. Quellen)
466             if (not "https://" in str(l) and not "http://" in str(l) and
467                 not str(l)[0] == '#'):
468                 sp = str(l).split('/')
469                 sp.pop(0)
470 
471                 try:
472                     #filtert alle unerwünschten Links heraus
473                     if sp[0] == "wiki":
474                         tg = sp[1].split(':')
475                         exc = ["Wikipedia", "Portal", "Spezial", "Kategorie",
476                                "Datei", "Hilfe", "Diskussion"]
477                         if tg[0] not in exc:
478                             links.append(l)
479                 #häufiger Fehler, wenn bei einem Eintrag keine Links gefunden
480                 #werden. Führte zu Programmabbruch. Fehler wird abgefangen.
481                 except Exception:
482                     return []
483 
484         result = []
485         for l in links:
486             #baut URL aus Wikipedia-internen Links
487             flink = "http://de.wikipedia.org" + str(l)
488             if flink not in result:
489                 result.append(flink)
490 
491         result.pop() #entfernt Link zu sich selbst
492 
493         return result
```

Abbildung 10: Code der Funktion getLinks(), eigene Darstellung

Im Konstruktor der Klasse `Eintrag` wird ebenfalls der Titel des aktuellen Eintrages über einen `print`-Befehl ausgegeben, sodass die Arbeit des Programms in Echtzeit über die Konsole nachverfolgt werden kann (s. Abbildung 11).

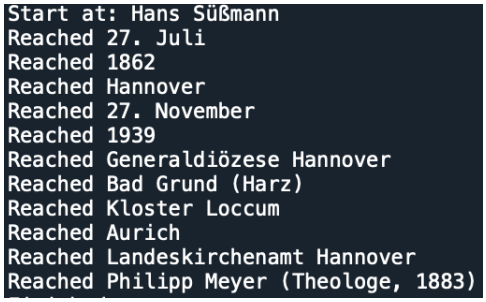

Abbildung 11: Beispiel einer Konsolenausgabe während des Indexierungsprozesses, eigene Darstellung

Persistente Datenspeicherung Auch der dritte Arbeitsschritt findet im Konstruktor statt. Dort wird die Funktion `addToGraph()` aufgerufen. Diese verwendet wiederum Funktionen der Bibliothek `networkX`, um dem Graphen, der als globale Variable gespeichert ist, einen neuen Knoten mit der URL des aktuellen Eintrags hinzuzufügen und in einem zweiten Schritt eine Kante von dem Knoten mit der URL, die im `source`-Attribut des Objekts gespeichert ist, zu dem eben erstellten Knoten mit der URL des aktuellen Eintrags legt. So wird der Eintrag dem Graphen als Knoten hinzugefügt und mit dem restlichen Graphen über eine gerichtete Kante verbunden. Bei Abschluss des Programms wird die Funktion `finishGraph()` aufgerufen, die den Graphen im `GraphML`-Format als `xml`-Datei speichert, sodass auch nachträglich Untersuchungen am Graphen angestellt werden können.

4 Repräsentation von Wikipedia durch einen Graphen

4.1 Zusammensetzung des Internets aus Graphen

Das Internet als Ganzes kann man sich als gerichteten Graphen vorstellen. Dabei setzt sich dieser Graph aus vier Teilgraphen zusammen. Das SCC bildet den Kern, ein Netzwerk aus großen Webseiten, die untereinander sehr stark vernetzt sind und sich durch einen besonders hohen Knotengrad auszeichnen. Zwei weitere Teilgraphen sind das IN und OUT, wobei das IN aus Webseiten besteht, die das SCC erreichen können, aber nicht vom SCC aus erreicht werden können. Das OUT dagegen ist ein Netzwerk von Webseiten, die vom SCC ausgehend erreicht werden können, dafür aber keine Webseiten des SCC finden können. Der vierte Teilgraph ist das sogenannte TENDRILS. Diese Webseiten können weder das SCC erreichen, noch vom SCC ausgehend

erreicht werden [Bro+00, S. 10 ff.].
Abbildung 12 zeigt den Aufbau als Schaubild.

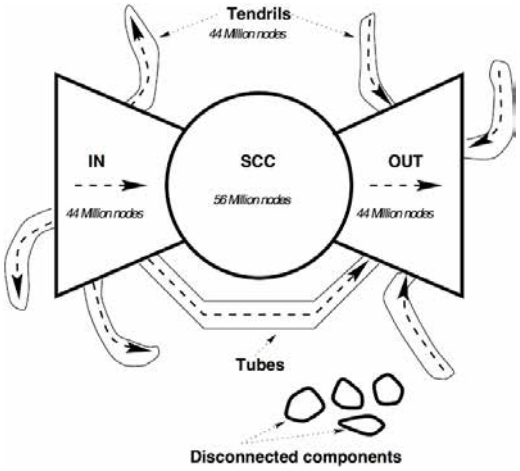

Abbildung 12: Schaubild der Graphenstruktur des Internets [Bro+00, S. 11]

4.2 Ansprüche an ein zu indexierendes Netzwerk

Der eben beschriebene Aufbau macht eine Untersuchung des gesamten World-Wide-Web überaus umständlich, da es schnell passieren kann, dass eine Webseite aus OUT oder TENDRILS in die Warteschlange des Webcrawlers gelangt und dieser deswegen nur einen kleinen Teil des Internets erreichen kann.

Eine weitere Schwäche des Webcrawlers hat vor allem in den letzten Jahren mit der Veränderung der Natur vieler Webseiten an Bedeutung gewonnen. Als die meisten einfachen Webcrawler entwickelt wurden, bestand das Internet vorwiegend aus statischen Webseiten, die lediglich im HTML-Code festgelegte Informationen darstellen. Mit der Zeit sind aber immer mehr Webseiten dynamisch geworden, sodass die dargestellten Informationen beim Laden der Webseite dynamisch und individuell generiert werden [YMH02, S. 2]. So sieht ein Nutzer, der sich mit seinem Benutzerkonto auf seinem Online-Banking-Portal anmeldet einen anderen Kontostand als einer seiner Freunde, oder die Wetter-App zeigt auf einem Smartphone in München ein anderes Wetter an, als die selbe App auf einem Smartphone in New York. Der HTML-Code für diese Seiten wird abhängig von bestimmten Umständen, wie dem angemeldeten Benutzerkonto oder der geographischen Position des Geräts individuell generiert. Das ist ein gravierendes Problem für Webcrawler, da diese keine Formulare ausfüllen können, was dazu führt, dass ihnen ein immer größer werdender Teil des Internets verschlossen bleibt [YMH02, S. 2].

Dies stellt diese Arbeit vor die Herausforderung eine vernünftige Wahl bezüglich der zu untersuchenden Webseiten zu treffen. Stark hierarchische Strukturen sind suboptimal, da sich der Webcrawler schnell in den tieferen Ebenen dieser „verlaufen“ kann und so nicht an bedeutende SCC-Seiten gelangt. Dynamische Webseiten sind wegen des erheblichen Aufwandes, der mit der Umprogrammierung des Webcrawlers zur Navigation dynamischer Websites verbunden wäre ebenfalls ausgeschlossen.
Die Anforderungen an ein Netzwerk zur Untersuchung sind also

- ausreichende Größe für bedeutende graphentheoretische Untersuchungen
- Zusammensetzung aus vorwiegend oder ausschließlich statischen Webseiten
- keine dominante hierarchische Struktur

Ein weniger bedeutender, aber dennoch nützlicher Vorteil wäre eine Zusammensetzung des Netzwerks aus etablierten und professionell kurierten Webseiten, da bei diesen Fehler, wie vergessene Hyperlinks seltener auftreten.

Wikipedia erfüllt nicht nur alle genannten Bedingungen, sondern eröffnet auch vollkommen neue Möglichkeiten, die bei anderen Netzwerken entweder unmöglich oder sinnlos wären. Durch den enzyklopädischen Fokus der Plattform können nicht nur strukturelle Untersuchungen angestellt werden, es ist ein vollkommen neuer und umfangreicher inhaltlicher Aspekt dazugekommen.

5 Untersuchung des Graphen

5.1 Anzahl der gefundenen Artikel in Abhängigkeit von der Untersuchungstiefe

Erstes Objekt der Untersuchungen ist die absolute Anzahl von, bei einer Indexierung gefundenen, Webseiten in Abhängigkeit von der rekursiven Untersuchungstiefe. Selbstverständlich wird die Anzahl gefundener Webseiten mit jeder zusätzlichen Rekursionsebene zunehmen, hier soll aber untersucht werden, ob sich diese Zunahme mathematisch durch eine Funktion beschreiben lässt und ob bei Variation der Startwebseite Unterschiede bei dem Wachstumsverhalten beobachtet werden können. Eine inhaltliche Beziehung zwi-

schen Startknoten und Anzahl gefundener Webseiten wäre ebenfalls von Interesse.

Durchführung Für diese Untersuchung sind keine komplexen Graphalgorithmen von Nöten. Der Webcrawler wird von einem festen Startknoten ausgehend eine Indexierung mit zunehmenden Untersuchungsebenen starten. So wird beim ersten Durchlauf eine Ebene durchsucht, beim zweiten zwei Ebenen, beim dritten drei Ebenen, etc.. Jeder Durchlauf startet vom selben Knoten aus, um Verfälschungen durch natürliche Unterschiede zu vermeiden.
Die Methode `testNodeIncrease()` ist eine praktische Implementierung des beschriebenen Versuchs. Dieser wird mehrmals ausgeführt und die Ergebnisse werden in einer Textdatei gespeichert.

Versuchsnr.	1 Ebene	2 Ebenen
1	150	9.635
2	55	6.849
3	70	6.288
4	49	5.015
5	108	11.248
6	136	11.015
7	47	9.065
8	27	9.624
9	35	10.891
10	8	4.010
∅	68,5	8.364

Tabelle 1: Ergebnisse der quantitativen Untersuchung gefundener Artikel

Ergebnisse und Interpretation Die Ergebnisse von zehn Versuchsdurchläufen sind in Tabelle 1 dargestellt.
Bei Betrachtung der Ergebnisse erkennt man, dass ein inhaltlicher Zusammenhang zwischen Artikel und Anzahl der Hyperlinks nur in der ersten Durchsuchungsebene bestehen kann. Dort ist er auch offensichtlich, da bedeutende Artikel mit hoher Wortzahl zwangsläufig mehr Hyperlinks aufweisen als unbedeutendere und weniger umfangreiche Artikel. So ist es nur logisch, dass ein Artikel, der sich mit allen Ereignissen eines Jahres beschäftigt mit mehr anderen Artikeln verlinkt ist als die Biographie eines unbekannten Musikers. Bereits in der zweiten Durchsuchungsebene verschwimmt der Zusammenhang, bei manchen Artikeln zeichnet sich der hohe Wert in Ebene 1 auf den Wert in Ebene 2 ab, so wie es in Versuch 1 zu sehen ist. Aber auch

hier gibt es Fälle, in denen Artikel, die in der ersten Durchsuchungsebene nur wenige Links hatten in der zweiten Ebene genauso viele oder sogar mehr Artikel aufweisen, als die mit vielen Links in der ersten Durchsuchungsebene. Es ist zu erwarten, dass dieser Zusammenhang mit jeder zusätzlichen Durchsuchungsebene immer weiter verschwindet.

Aus den Durchschnittswerten der Ergebnisse wurde die Exponentialfunktion

$$f(x) = 0,6 \cdot 120^x$$

formuliert, die die absolute Anzahl von gefundenen Artikeln in Abhängigkeit von der Anzahl der Durchsuchungsebenen zeigt.

Diese Funktion liefert einige interessante Informationen. So lässt sich ablesen, dass ein Wikipedia-Artikel im Durchschnitt mit 120 anderen Artikeln verknüpft ist. Außerdem kann man einen Näherungswert für die Anzahl der gefundenen Artikel in höheren Durchsuchungsebenen berechnen. Dabei zeigt sich, dass man bei drei Ebenen bereits auf über eine Millionen Artikel kommt, während es bei zwei Ebenen nur etwa 9000 sind. Das heißt, dass man nach nur drei Untersuchungsebenen bereits fast die Hälfte der gesamten deutschsprachigen Wikipedia indexiert hat. Laut Wikipedia selbst besteht diese nämlich aus etwa 2,6 Millionen Artikeln [Wik21a]. Die gesamte Wikipedia mit 52 Millionen Artikeln in 300 Sprachen [Wik21b] hätte man mit nur einer zusätzlichen Durchsuchungsebene abgedeckt.
Das sind natürlich nur theoretische Werte, in der Realität bilden die verschieden Sprachen eine Barriere, die den Aufwand enorm erhöht und auch die Vernetzungsdichte variiert stark zwischen verschiedenen Artikeln und Sprachen. Dennoch kann man sich so ein Bild machen, wie mächtig die simplen Hyperlinks Wikipedias sind und auch wie gut die gesamte Plattform strukturiert ist. Die Indexierung von 120 Millionen Artikeln mit nur 4 Durchsuchungsebenen ist nur durch Artikel, wie „6. Januar “ oder „20. Jahrhundert“ möglich. Artikel wie diese verbinden enorm viele Artikel, die sich ein gemeinsames Datum teilen, miteinander, was eine Indexierung mit so wenigen Schritten ermöglicht.

Probleme Beim erstmaligen Start des Versuchs haben sich einige Probleme gezeigt. Das erste und offensichtlichste Problem ist die hohe Laufzeit des Programms. Um die Untersuchung durchzuführen muss wiederholt ein immer größer werdender Teil Wikipedias indexiert werden, was bei einer einzigen Untersuchungsebene kein wirkliches Problem ist, aber schon in der zweiten Ebene steigt die Anzahl der Artikel enorm. Dazu kommt, dass der

Versuch mehrere Male mit verschiedenen Startartikeln wiederholt werden muss, eine Aufgabe, die mit der aktuellen Webcrawler-Architektur und der zur Verfügung stehenden Rechenleistung schlichtweg nicht zu bewältigen ist. Um dennoch an aussagekräftige Daten zu kommen, wurde der Versuch parallel auf fünf Computer durchgeführt, was einige Ergebnisse für die erste und zweite Durchsuchungsebene geliefert hat. Aber selbst nach 24 Stunden hat keiner der fünf Rechner die dritte Durchsuchungsebene beendet. Der Versuch konnte auch nicht bedeutend verlängert werden, da immer häufiger Verbindungsfehler aufgetreten sind oder dem Computer nicht mehr genügend Arbeitsspeicher zur Verfügung stand. Deswegen beschränken sich die in Tabelle 1 aufgeführten Daten nur auf die erste und zweite Durchsuchungsebene, Werte für höhere Ebenen sind nur durch die Funktion $f(x) = 0,6 \cdot 120^x$ berechnete Näherungswerte.

Optimierung Eine mögliche Optimierung könnte die Laufzeit des Versuchs verkürzen, ohne die grundlegende Architektur des Webcrawlers zu verändern.
Statt die Indexierung jedes Mal von dem selben Artikel ausgehend zu starten und bei jedem Durchlauf die Durchsuchungsebenen um den Wert 1 zu erhöhen, könnten die Endknoten von jedem Durchlauf gespeichert werden, sodass bei der nächsten Iteration nur eine Indexierung über eine Ebene von jedem Endknoten ausgehend gestartet werden muss. Die Zeitersparnis dabei bezieht sich nur auf die Wiederholung der bereits indexierten Ebenen. Bei nur einer Durchsuchungsebene ergibt sich dabei keine Zeitersparnis, bei der dritten Ebene beträgt diese allerdings schon etwa eine Stunde. Die Zeitersparnis für höhere Ebenen kann nicht vorhergesagt werden, da die tatsächliche Laufzeit einer gesamten Indexierung unbekannt ist, wobei sie in jedem Fall über 40 Stunden liegt.
Nach der Implementierung dieser Optimierung hat sich gezeigt, dass keine bedeutende Laufzeitverbesserung festzustellen ist. Die Wiederholung bereits indexierter Bestandteile scheint also keinen wirklichen Einfluss auf die Gesamtlaufzeit des Programms zu haben, die stark anwachsende Menge an Artikeln fällt deutlich schwerer ins Gewicht.

Die Optimierung, die den wahrscheinlich größten Einfluss auf die Laufzeit hätte, wäre eine an Parallelisierung orientierte Implementierung. Dabei wird die Arbeitslast entweder auf jeden einzelnen Rechenkern des Prozessors oder sogar auf mehrere Rechner aufgeteilt, um ein Problem schneller zu lösen. Eine solche Implementierung erfordert allerdings umfassende Kenntnisse der

Informatik, die zum Zeitpunkt der Verfassung dieser Arbeit nicht angeeignet wurden.

5.2 Existenz und Anzahl von Zyklen

Als nächstes soll der Graph auf Zyklen untersucht werden. Diese für sich sind inhaltlich von keinem großen Interesse, da zu erwarten ist, dass unmittelbar miteinander verknüpfte Artikel auch einen inhaltlichen Zusammenhang aufweisen. Allerdings ist das Vorhandensein von Zyklen eine Grundbedingung für die nächste Untersuchung, den kürzesten Weg zwischen zwei, bei erster Betrachtung, beziehungslosen Artikeln. Außerdem können durch die Anzahl der Zyklen in einem Graphen bedeutende Aussagen über die Vernetzungsdichte der Struktur getroffen werden.

Durchführung `networkX` hat für die Suche von Zyklen einige Funktionen, die hier auch zum Einsatz gekommen sind. Die Untersuchungen wurden an einem gerichteten Graphen ausgeführt, der aus etwa 36.000 Knoten besteht, die durch eine Indexierung auf zwei Ebenen entstanden sind. Zunächst wurde die Funktion `simple_cycles(G)`, wobei `G` ein gerichteter Graph ist, verwendet, die jeden, nicht-trivialen Zyklus als Liste von Knoten als Rückgabewert hat. Die Länge dieser Liste wurde über einen `print`-Befehl ausgegeben.

Ergebnisse und Interpretation In dem gesamten Graphen aus 36.000 Knoten war kein einziger Zyklus zu finden. Manuelle Untersuchungen haben gezeigt, dass das Ergebnis nicht stimmen kann, da es zweifelsfrei Zyklen in Wikipedia gibt. Dabei hat sich aber eine wichtige Bedingung gezeigt: ein nicht-trivialer Zyklus besteht aus mindestens drei Knoten, die so durch Kanten verbunden sind, dass man vom Starknoten ausgehend diesen wieder erreichen kann (vgl. 2.2, „Zyklus“). Bei ungerichteten Graphen wäre diese Bedingung nicht weiter problematisch, aber für den gerichteten Graphen heißt das, dass bei zwei Untersuchungsebenen bei der Indexierung unmöglich ein nicht-trivialer Zyklus enthalten sein kann.

Optimierung Nach dieser Erkenntnis wurde offensichtlich, dass die weiteren Untersuchungen keinesfalls am gerichteten Graphen ausgeführt werden können.
Dies ist aber kein gravierendes Problem, da die Funktion `to_undirected()` aus einem gerichteten Graphen in Sekundenbruchteilen einen ungerichteten macht. In dem ungerichteten Graphen kann mit der Funktion `cycle_basis()` nach Zyklen gesucht werden. Mit dieser Methode wurden auch etwa 31.000

Zyklen gefunden, was auf die erwartete enorm hohe Vernetzungsdichte schließen lässt. Diese ist in Abbildung 13 und den Abbildungen von Graphen im Anhang visualisiert dargestellt. Dabei stellen die blauen Punkte Knoten dar, die schwarzen Flächen dazwischen sind sich überschneidende Kanten, eine Differenzierung dieser ist selbst bei Abbildungen in größeren Formaten aufgrund der enormen Anzahl nicht möglich. Im Anhang sind Abbildungen von kleineren Graphen zu finden, bei denen sich der Verlauf der einzelnen Kanten besser beobachten lässt.

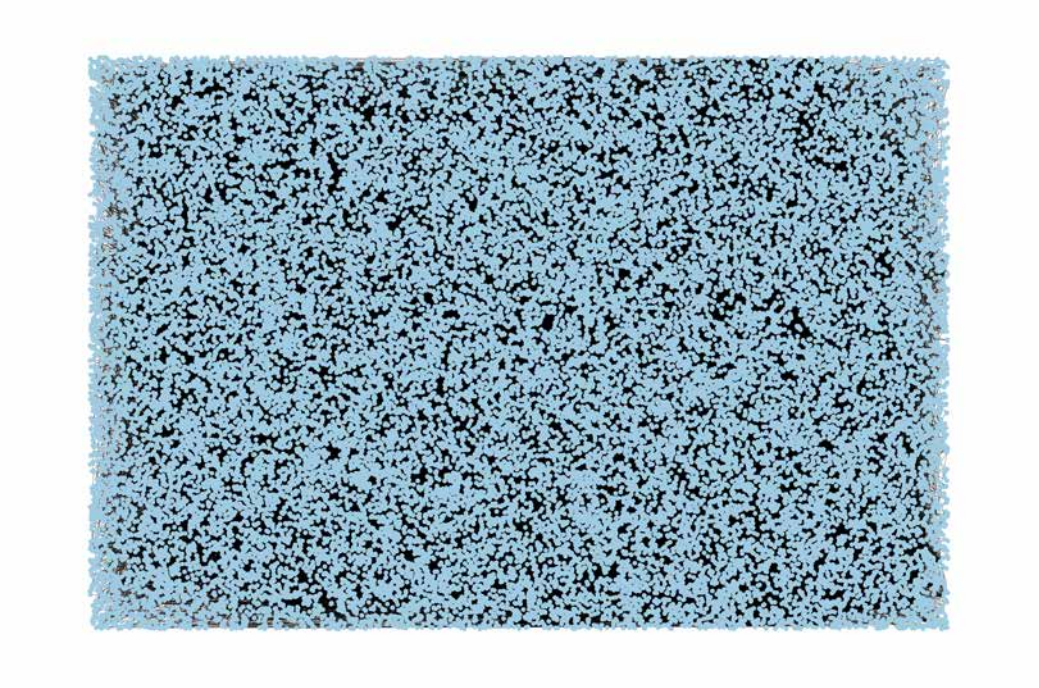

Abbildung 13: Visualisierung eines Graphen aus ca. 36.000 Knoten, eigene Darstellung

5.3 Kürzester Weg zwischen beziehungslosen Artikeln

Der vorherige Versuch hat gezeigt, dass diese Untersuchung nicht in der geplanten Art und Weise stattfinden kann. Die Idee war, dass der krüzeste Weg zwischen zwei zufälligen Artikeln des Graphen gesucht wird, was wiederum eine Untersuchung des inhaltlichen Zusammenhangs der beiden Artikel ermöglicht. Durch erste Versuche hat sich schnell herausgestellt, dass die Untersuchung zweier zufälliger Knoten keine Option ist. Da der Graph gerichtet und durch Indexierung von nur 2 Ebenen entstanden ist, kann ein Pfad sich höchstens über drei Knoten erstrecken, was nicht genug Raum für

die Entdeckung neuer inhaltlicher Zusammenhänge bietet. Auch ein Pfad über drei Knoten ist nur möglich, wenn der Startknoten des Pfads gleichzeitig der Startknoten bei der Indexierung war. Dafür müsste bei jedem zu untersuchenden Artikel eine neue Indexierung gestartet werden, ein Vorgehen, das aufgrund der Laufzeit des Programms nicht sinnvoll wäre.
Die Alternative ist das in 5.2 bereits verwendete Vorgehen. Durch Entfernen der Richtung der Kanten eröffnen sich neue Möglichkeiten, da in einem ungerichteten, zusammenhängenden Graphen jeder Knoten von jedem beliebigen Startknoten ausgehend erreicht werden kann. Dabei geht allerdings ein Aspekt verloren. Bei der Untersuchung des gerichteten Graphen können die ausgegebenen Pfade auch in der Realität nachvollzogen und in der selben Reihenfolge in Wikipedia abgearbeitet werden. Dies ist nach der Umstellung von gerichtet auf ungerichtet nicht immer möglich. In Anbetracht der Tatsache, dass primäres Ziel des Versuchs das Auffinden von bisher unbekannten Zusammenhängen ist, lässt sich dieser Umstand vernachlässigen.

Durchführung Um den Versuch durchzuführen, wurde eine Funktion `testShortestPath()` implementiert, die zunächst den gerichteten Graphen ungerichtet macht. Daraufhin wird der kürzeste Weg zwischen zwei zufälligen Knoten gesucht und der gesamte Pfad wird in einer Textdatei gespeichert. So kann man in der Textdatei nach besonders interessanten Zusammenhängen suchen und diese genauer begutachten. Der Versuch wird an einem ungerichteten Graphen aus etwa 35.000 Knoten durchgeführt.

Ergebnisse und Interpretation Die Ergebnisse des Versuches sind nicht wie erwartet ausgefallen. Es hat sich gezeigt, dass dieses Vorgehen ungeeignet zum Auffinden neuer Zusammenhänge ist, was vor allem auf die zuvor gelobte Struktur Wikipedias zurückzuführen ist.

Während große Artikel, die einen weiten Bereich abdecken, wie zum Beispiel die von Jahreszahlen, alle Artikel sehr eng miteinander vernetzen und vorbildlich ordnen, kommt diese Ordnung nur in Kombination mit dem Verlust jeglicher realer inhaltlicher Bedeutung der Verbindungen. Artikel, wie diese sind der Grund dafür, dass eine Kante im Graphen nicht zwangsläufig bedeutet, dass die Knoten, die sie verbindet, einen wirklichen inhaltlichen Zusammenhang haben. Denn obwohl die Heirat einer Person im selben Jahr stattgefunden hat, wie die Anmeldung eines Patentes auf der anderen Seite des Planeten, heißt das nicht, dass die beiden Ereignisse in irgendeiner inhaltlichen Beziehung zueinander stehen. Dieser Verlust des Zusammenhangs wird durch das Entfernen der Richtung der Kanten nur zusätzlich verstärkt.

Besonders ins Auge sticht hierbei der Artikel über das „Internet Archive“, zu dem ein Link in den Einzelnachweisen von nahezu jeder Person zu finden ist, was ihn zu einem wichtigen „Drehkreuz“ bei der Suche kürzester Wege macht.

Eine weitere Verfälschung entsteht durch den vergleichsweise kleinen Anteil Wikipedias der indexiert wurde. Der Graph repräsentiert nur etwa 1,3% der deutschen Wikipedia, was zu sehr einschlägigen Ergebnissen führt. Ist die Person, von deren Wikipedia-Eintrag die Indexierung gestartet ist, im Jahr 1965 geboren, so fällt auf, dass sich ein Großteil der gefundenen Pfade über den Knoten „1965“ erstreckt. Ein weiterer wichtiger Knotenpunkt im Testgraphen war der Eintrag über die Serie „Alarm für Cobra 11 – Die Autobahnpolizei“, an der die Person, von der ausgehend die Indexierung gestartet ist, mitgearbeitet hat. Bleibt die Indexierung bei nur zwei Ebenen, so wird es immer solche Knotenpunkte geben, die zwar eine reale Verbindung zum Startknoten haben, aber auch andere Knoten miteinander verbinden, ohne einen wirklichen Zusammenhang aufweisen zu können.

Optimierung Diese Schwächen könnten vermieden werden, indem man nicht immer nach dem kürzesten Weg zwischen den Knoten sucht, sondern sich jede mögliche Verbindung zwischen zwei Knoten ausgeben lässt. Hier liegt das Problem wiederum in der Graphentheorie, die besagt, dass es in einem gerichteten, zyklischen Graphen unendlich viele Pfade zwischen zwei Knoten v_n und v_m gibt. Bei der Implementierung ließe sich das umgehen, indem man festlegt, dass jeder Knoten nur ein einziges Mal besucht werden darf, aber bei ausreichend großen Graphen würde die Menge möglicher Pfade auch hier die Menge überschreiten, die von Menschen verarbeitet werden könnte.
Der kürzeste Weg könnte bei einer vollständigen Indexierung Wikipedias von Bedeutung sein, wieso diese hier aber nicht möglich ist wurde im Abschnitt „Optimierung“ des Kapitels 5.1 bereits erläutert.
Eine Lösung für dieses Problem läge wiederum in der künstlichen Intelligenz, die aber nicht Teil dieser Arbeit sein soll.

Der kürzeste Weg könnte bei einer vollständigen Indexierung Wikipedias von eine reale inhaltliche Bedeutung haben, wieso diese hier aber nicht möglich ist wurde im Abschnitt „Optimierung“ des Kapitels 5.1 bereits erläutert.

6 Fazit

Beginnend bei dem Webcrawler ist zu sagen, dass die Arbeitslast und die schiere Größe der Aufgaben massiv unterschätzt wurden. Zahlen, wie die 2,6 Millionen Artikel der deutschen Wikipedia [wel16] verlieren im Zusammenhang mit Computern schnell an Bedeutung, allerdings hat sich deren wahres Ausmaß vor allem durch den in 5.1 beschriebenen Versuch gezeigt. Dieser Versuch hat sich vor allem im Gesamtbild als besonderes problematisch gezeigt, was in erster Linie auf eine bewusst suboptimale Implementierung zurückzuführen ist. Diese entstand vor dem Hintergrund, dass die Arbeitslast auf den selbst programmierten Webcrawler nicht mit der kommerzieller Webcrawler zu vergleichen ist und deswegen Abstriche bei der Laufzeitoptimierung möglich sind. Es hat sich gezeigt, dass dieser Umstand eine vollständige und aussagekräftigere Indexierung verhindert hat. Somit hat sich gezeigt, dass jede Optimierungsmöglichkeit verfolgt werden sollte, da die tatsächliche Arbeitslast oft nicht intuitiv abgeschätzt werden kann, vor allem wenn es um rekursive Prozesse geht. Aber auch dies sollte nur in vernünftigen Maßen geschehen, die Implementierung eines Taschenrechners mit maschinellem Lernen würde sich nicht als besonders sinnvoll erweisen.

Zur Struktur Wikipedias lässt sich sagen, dass sie bemerkenswert gut organisiert und dadurch schnell zu navigieren ist. Die Anzahl gefundener Artikel in Abhängigkeit von der Anzahl der Durchsuchungsebene lässt sich durch die Funktion $f(x) = 0,6 \cdot 120^x$ näherungsweise beschreiben.
Wikipedia ist ein gerichteter, zyklischer Graph, was allerdings wegen der geringen Anzahl von Durchsuchungsebenen nicht Teil der Untersuchungen sein konnte, sich aber durch manuelle Tests gezeigt hat. Eine Abwandlung der realen Struktur hat weitere Untersuchung ermöglicht, die eine sehr hohe Vernetzungsdichte Wikipedias aufzeigen, eine Erkenntnis, die durch Visualisierung der Graphen gestützt wird.
Durch Einträge, die ganze Listen von Ereignissen und Daten enthalten, ist es möglich, dass man einen enorm großen Teil Wikipedias mit nur wenigen Untersuchungsebenen indexieren kann. Dies ermöglicht eine schnelle Navigation zwischen Artikeln sehr unterschiedlichen Inhalts, was allerdings mit dem Verlust der inhaltlichen Bedeutung von Hyperlinks verbunden ist. So kann man nur durch das Vorhandensein eines Hyperlinks zu einem Wikipedia-Artikel nicht unbedingt von einem realen Zusammenhang ausgehen.

Bei den Untersuchungen zur Struktur sind teilweise überaus spektakuläre Bilder entstanden, eine Auswahl dieser ist im Anhang zu finden.

A Vollständiger Quellcode des Programms

```python
# -*- coding: utf-8 -*-
import requests
from bs4 import BeautifulSoup
from requests.utils import quote
import networkx as nx
import os.path
import matplotlib.pyplot as plt
from random import sample


"""
Created on Tue Feb  9 13:52:10 2021
Ein Webcrawler, der Wikipedia-Einträge indexieren soll, sodass diese im
Nachhinein auf strukturelle und inhaltliche Besonderheiten untersucht werden
können.
@author: Faris Avdic
"""


class Webcrawler:
    """
    Eine Klasse, die einen Webcrawler implementiert. Der Webcrawler startet
    die Indexierung und führt diverse Tests aus.

    Attribute
    ---------
    graph: networkx.DiGraph
        Der Graph, der bei der Indexierung erstellt wird.

    Methoden
    --------
    initializeGraph()
        initialisiert das Attribut graph
    finishGraph()
        speichert den Graphen als xml-Datei im GraphML-Format
    resetGraph()
        setzt den Graphen zurück
    start(url,layers)
        startet eine Indexierung von url mit layers Ebenen
    getRandomURL()
        gibt die URL zu einem zufälligen Wikipedia-Artikel zurück
    createGraph(layers)
        ruft start() auf, fängt evtl. auftretende Fehler ab
    drawGraph()
        zeichnet den Graphen
    saveGraphImage()
        speichert das Bild des Graphen als png-Datei
    testNodeIncrease(runs,layers)
        Führt den Test aus Kapitel 5.1 der Seminararbeit durch
    testShortestPath()
        Führt den Test aus Kapitel 5.3 der Seminararbeit durch
    testCycles()
        Führt den Test aus Kapitel 5.2 der Seminararbeit durch
    """
```

```
def __init__(self):
    """
    Konstruktor der Klasse Webcrawler.

    Rückgabe
    --------
    None
    """

    self.graph = self.initializeGraph()


def initializeGraph(self):
    """
    Initialisiert das Attribut graph des Webcrawlers. Wenn eine
    Datei "graph.xml" existiert, wird diese eingelesen und dem Attribut
    zugeordnet.
    Existiert die Datei nicht, wird dem Attribut ein neues Objekt der
    Klasse networkx.DiGraph zugeordnet.

    Rückgabe
    --------
    tempGraph : networkx.DiGraph
        Gerichteter Graph
    """

    if os.path.isfile("graph.xml"):
        tempGraph = nx.read_graphml("graph.xml")
        print("Graph exists and is read")
    else:
        tempGraph = nx.DiGraph()
        print("Graph doesn't exist and has been created")
    print("Initialized.")
    return tempGraph


def finishGraph(self):
    """
    Schleißt die Arbeit am Graphen ab, indem die Anzahl der Knoten über
    die Konsole ausgegeben wird und der Graph im GraphML-Format
    als xml-Datei gespeichert wird.

    Returns
    -------
    None
    """

    print("The graph consists of " + str(
        self.graph.number_of_nodes()) + " nodes.")
    nx.write_graphml(self.graph,"graph.xml")
    print("Finished.")
```

```
def resetGraph(self):
    """
    Setzt den Graphen zurück, indem die Datei "graph.xml" gelöscht wird.

    Rückgabe
    --------
    None
    """

    try:
        os.remove("graph.xml")
        print("Graph reset successful.")
    except Exception as e:
        print("Graph reset failed.")
        print(e)


def start(self, url, layers):
    """
    Startet die Indexierung.

    Parameter
    ---------
    url : str
        Die URL des Eintrags, bei dem die Indexierung beginnen soll.
    layers : int
        Anzahl der Ebenen, in denen die Indexierung durchgeführt werden
        soll.
        layers = 0 -> Nur die Links, die im Start-Eintrag gefunden werden,
            werden indexiert.
        layers = 1 -> Start-Eintrag, direkt verlinkte Einträge und alle
            dort gefunden Links werden indexiert.

    Rückgabe
    --------
    res : list
        Vielfach verschachtelte Liste, die alle gefundenen Einträge
        beinhaltet.
        In Baumstruktur umwandelbar.
    """
    startEntry = Eintrag(url, url, self)

    print("Start at: " + startEntry.title)
    res = startEntry.index(layers,self)
    print("Finished")

    return res


def getRandomURL(self):
    """
    Gibt die URL zu einem zufälligen Wikipedia-Eintrag zurück.
    Die Funktion nutzt die "Zufälliger Artikel"-Funktion von Wikipedia.
```

```
        Rückgabe
        --------
        url : str
            URL zu einem zufälligen Wikipedia-Eintrag
        """
        res = requests.get(
            "https://de.wikipedia.org/wiki/Spezial:Zuf%C3%A4llige_Seite").text
        soup = BeautifulSoup(res, 'html.parser')
        title = soup.find('title').text
        newTitle = title.replace(' – Wikipedia', '')
        urlText = quote(newTitle, safe='')
        url = "http://de.wikipedia.org/wiki/" + urlText
        return url


    def createGraph(self,layers):
        """
        Startet eine Indexierung über layers Ebenen.
        Ruft nur start() auf, schließt Methode aber in eine
        try-except-Anweisung ein, was ermöglicht, dass das Programm auch bei
        Fehlermeldungen normal beendet werden kann und nicht der gesamte
        Fortschritt der Indexierung verloren geht.

        Parameters
        ----------
        layers : int
            Anzahl der Untersuchungsebenen bei der Indexierung

        Rückgabe
        --------
        None
        """

        try:
            self.start(self.getRandomURL(), layers)
            print("SUCCESSFULLY FINISHED RUN.")
        except Exception as e:
            print("AN ERROR HAS OCCURED.")
            print(e)


    def drawGraph(self):
        """
        Zeichnet ein Bild des Graphen mit dem "random"-Layout.
        Bild wird im "Plots"-Fenster von Spyder angezeigt.

        Rückgabe
        --------
        None
        """

        print("Generating layout.")
        pos = nx.random_layout(self.graph)
        print("Finished generating layout.")
```

```
        print("Drawing graph.")
        nx.draw(self.graph,pos,node_size=1,width=0.1,
                node_color="#A0CBE2",with_labels=False,arrowsize=1)
        print("Finished drawing graph.")


    def saveGraphImage(self):
        """
        Speichert das Bild des Graphen als png-Datei.

        Rückgabe
        --------
        None
        """

        print("Saving image to file.")
        plt.savefig("graph.png", dpi =2000)
        print("Finished saving image to file.")


    def testNodeIncrease(self, runs, layers):
        """
        Untersucht den Graph auf sein Wachstumsverhalten bei steigender
        Anzahl von Untersuchungsebenen.
        Die Indexierung startet von einem Artikel ausgehend mit nur einen
        Durchsuchungsebene. Nach diesem ersten Durchlauf wird von jedem der
        Endknoten des Graphen (Grad 1) eine neue Indexierug über eine
        Durchsuchungsebene gestartet. Der Prozess wird so lange wiederholt,
        bis insgesamt layers Ebenen durchsucht wurden.
        Die Anzahl der Knoten, aus denen der Graph besteht wird nach jeder
        Iteration in einer Textdatei gespeichert und über die Konsole
        ausgegeben.

        Parameter
        ---------
        runs : int
            Anzahl der Wiederholungen des Versuchs
        layers : int
            Anzahl der maximalen Durchsuchungsebenen bei der Indexierung

        Rückgabe
        --------
        None
        """

        self.resetGraph()
        for i in range(runs):
            url = self.getRandomURL()
            self.start(url, 0)
            string=str(i)+".0: "+str(self.graph.number_of_nodes())+"\n"
            print(string)
            with open("testNodeIncrease.txt","a") as file:
                file.write(string)
```

```
        for c in range(layers):
            nodeList=self.graph.nodes()
            endList=[]
            for n in nodeList:
                if self.graph.degree(n) == 1:
                    endList.append(n)
            for e in endList:
                self.start(e,0)
            string=str(i)+"."+str(c)+": "+str(
                self.graph.number_of_nodes())
            print(string)
            with open("testNodeIncrease.txt","a") as file:
                file.write(string)


def testShortestPath(self):
    """
    Erstellt eine ungerichtete Kopie des Graphen aus dem Attribut
    graph.
    Sucht den kürzesten Weg zwischen zwei zufälligen Knoten im
    ungerichteten Graphen.
    Speichert jeden einzelnen Knoten des Pfades in einer Textdatei und
    gibt den Pfad über die Konsole aus.

    Rückgabe
    --------
    None
    """

    uGraph = nx.to_undirected(self.graph)

    randomNodes = sample(uGraph.nodes(),2)
    randomNode1 = str(randomNodes[0])
    randomNode2 = str(randomNodes[1])

    print("rand1: " + randomNode1)
    print("rand2: " + randomNode2)

    try:
        shortestPath = nx.shortest_path(uGraph, randomNode1, randomNode2)
        with open("paths.txt","a") as file:
            for i in shortestPath:
                print(i)
                file.write(i + "\n")
            file.write("-------------------------")
    except Exception as e:
        print(e)


def testCycles(self):
    """
    Macht aus einem gerichteten Graphen einen ungerichteten und sucht
    alle Zyklen im Graphen heraus.
    Gibt Anzahl der gefundenen Zyklen über die Konsole aus.
```

```
        Rückgabe
        --------
        int
            Anzahl der Zyklen im Graphen

        """
        uGraph = self.graph.to_undirected()
        cycleList = list(nx.cycle_basis(uGraph))
        print(len(cycleList))
        return len(cycleList)


class Eintrag:
    """
    Eine Klasse, die einen Wikipedia-Eintrag mit allen wichtigen Informationer
    repräsentiert.

    Attribute
    ---------
    url : str
        Die URL des Wikipedia-Eintrags als String.
        Wird verwendet um Einträge eindeutig zu identifizieren.
    source : str
        Die URL des Eintrags, in dem der Hyperlink zum aktuellen Eintrag
        gefunden wurde.
    html : str
        Der HTML-Code des Wikipedia-Eintrags.
    text : str
        Der Text auf der Wikipedia-Seite.
    title : str
        Der Titel des Wikipedia-Eintrags.
    links : list
        Eine Liste mit allen Hyperlinks, die im HTML-Code des Eintrags
        gefunden wurden.
        Die Liste beinhaltet nur die Links, die auch zu anderen
        Wikipedia-Einträgen führen.
        Diskussions- und Spezialseiten werden vorher herausgefiltert.
    leaves : list
        Eine Liste mit anderen Objekten der Klasse Eintrag.
        Speichert alle Einträge, zu denen Hyperlinks im Code des aktuellen
        Eintrags führen.
    crawler : Webcrawler
        Der Webcrawler, der die aktuelle Indexierung ausführt.
        Attribut nötig, um dem Graphen Knoten hinzufügen zu können.

    Methoden
    --------
    index(layers, crawler)
        Rekursive Methode, die Wikipedia-Einträge indexiert.
    getLinks()
        Sucht alle Links im HTML-Code des Eintrags und filtert unerwünschte
        Links heraus.
```

```
getHTMLCode()
    Gibt den HTML-Code des Eintrags zurück.
getText()
    Gibt den Text des Wikipedia-Eintrags zurück.
getTitle()
    Gibt den Titel des Wikipedia-Eintrags zurück.
addToGraph(crawler)
    Fügt den Eintrag zum Graphen hinzu.
"""

def __init__(self, url, source, crawler):
    """
    Konstruktor der Klasse Eintrag.

    Parameter
    ---------
    url : str
        Die URL des Wikipedia-Eintrags, der vom Objekt repräsentiert
        werden soll.
    source : str
        Die URL des Wikipedia-Eintrags, in dem der Link zum aktuellen
        Eintrag gefunden wurde.
    crawler : Webcrawler
        Objekt des Webcrawlers, der die aktuelle Indexierung ausführt.
        Parameter notwendig, um dem Graphen Knoten hinzufügen zu können.
    """
    self.url = url
    self.source = source
    self.html = self.getHTMLCode()
    self.text = self.getText()
    self.title = self.getTitle()
    print("Reached " + self.title) #ermöglicht Beobachtung des Programms
                                   #in Echtzeit über die Konsolenausgabe.
    self.links = []
    self.leaves = [] #speichert Referenzen auf andere Einträge
    self.crawler = crawler
    self.addToGraph(crawler)


def __repr__(self):
    return self.url


def index(self, layers, crawler):
    """
    Rekursive Methode, die Wikipedia-Einträge indexiert.

    Parameter
    ---------
    layers : int
        Zählvariable, die die Anzahl der übrigen, noch zu indexierenden,
        Ebenen speichert.
        Wird bei jedem rekursiven Durchlauf um 1 verkleinert.
    crawler : Webcrawler
```

```
            Objekt des Webcrawlers, der die Indexierung ausführt.
            Parameter notwendig, um Knoten dem Graphen hinzufügen zu können.

        Rückgabe
        --------
        self.leaves : list
            Gibt den Attributwert der Instanzvariable leaves zurück.
        """

        if layers < 0:
            return self
        else:
            self.links = self.getLinks()
            for l in self.links:
                nextE = Eintrag(l, self, crawler)
                self.leaves.append(nextE.index(layers - 1,self.crawler))
            return self.leaves


    def getLinks(self):
        """
        Findet alle Links im Artikel und filtert unerwünschte heraus.
        Unerwünscht sind alle Links, die nicht zu anderen Wikipedia-Einträgen,
        sondern zu bestimmten Spezialseiten führen.

        Rückgabe
        --------
        list
            Eine Liste, die alle erwünschten Links beinhaltet.

        """
        soup = BeautifulSoup(self.html, "html.parser")
        allLinks = soup.findAll('a', href=True)

        #alle URLs (href-Attribute der <a>-Tags) werden in Liste hrefs
        #gespeichert
        hrefs = []
        for l in allLinks:
            hrefs.append(l["href"])

        links = []
        for l in hrefs:
            #filtert Links heraus, die zu anderen Internetseiten führen
            #(nicht Wikipedia, z.B. Quellen)
            if (not "https://" in str(l) and not "http://" in str(l) and
                not str(l)[0] == '#'):
                sp = str(l).split('/')
                sp.pop(0)

                try:
                    #filtert alle unerwünschten Links heraus
                    if sp[0] == "wiki":
                        tg = sp[1].split(':')
                        exc = ["Wikipedia", "Portal", "Spezial", "Kategorie",
```

```
                                "Datei", "Hilfe", "Diskussion"]
                        if tg[0] not in exc:
                            links.append(l)
                #häufiger Fehler, wenn bei einem Eintrag keine Links gefunden
                #werden. Führte zu Programmabbruch. Fehler wird abgefangen.
                except Exception:
                    return []

        result = []
        for l in links:
            #baut URL aus Wikipedia-internen Links
            flink = "http://de.wikipedia.org" + str(l)
            if flink not in result:
                result.append(flink)

        result.pop() #entfernt Link zu sich selbst

        return result


    def addToGraph(self, crawler):
        """
        Fügt den Eintrag als Knoten zum networkX-Graph hinzu.
        Fügt Kante von source-Eintrag zu aktuellem Eintrag zum
        networkX-Graph hinzu.

        Eine Überprüfung auf doppelte Knoten ist nicht nötig, da networkX
        diese automatisch überschreibt.
        Da im Knoten nur die URL gespeichert wird, ist die Überschreibung mit
        identischen Werten kein Problem.
        WICHTIG: Kanten bleiben Bestehen, auch wenn der Knoten überschrieben
        wird.

        Parameter
        ---------
        crawler : Webcrawler
            Der Webcrawler, der die aktuelle Indexierung ausführt.
            Parameter nötig, um dem Graphen Knoten hinzufügen zu können.

        Rückgabe
        --------
        None
        """
        crawler.graph.add_node(self.url)
        crawler.graph.add_edge(self.source, self.url)


    def getHTMLCode(self):
        """
        Gibt den HTML-Code des aktuellen Eintrags als String zurück.

        Rückgabe
        --------
        str
```

```
            vollständiger HTML-Code des Eintrags
        """
        res = requests.get(self.url)
        return res.text


    def getText(self):
        """
        Gibt den Verfassertext des Wikipedia-Eintrags als String zurück.

        Rückgabe
        --------
        str
            Verfassertext des aktuellen Wikipedia-Eintrags
        """
        soup = BeautifulSoup(self.html, 'html.parser')
        return soup.get_text()


    def getTitle(self):
        """
        Gibt den Titel des aktuellen Wikipedia-Eintrags als String zurück.

        Rückgabe
        --------
        str
            Titel des Eintrags
        """
        soup = BeautifulSoup(self.html, 'html.parser')
        ft = soup.find("title").text
        t = ft.split(' ')
        t.pop()
        t.pop()
        return ' '.join(t)


#-----------------------------------------------------------------------
crawler = Webcrawler()

crawler.createGraph(0)

crawler.drawGraph()
crawler.saveGraphImage()

crawler.testNodeIncrease(1,2) #VORSICHT: LÖSCHT AKTUELL GESPEICHERTEN GRAPHEN
crawler.testCycles()
crawler.testShortestPath()

crawler.finishGraph()
```

B Visualisierte Abbildungen verschiedener Graphen

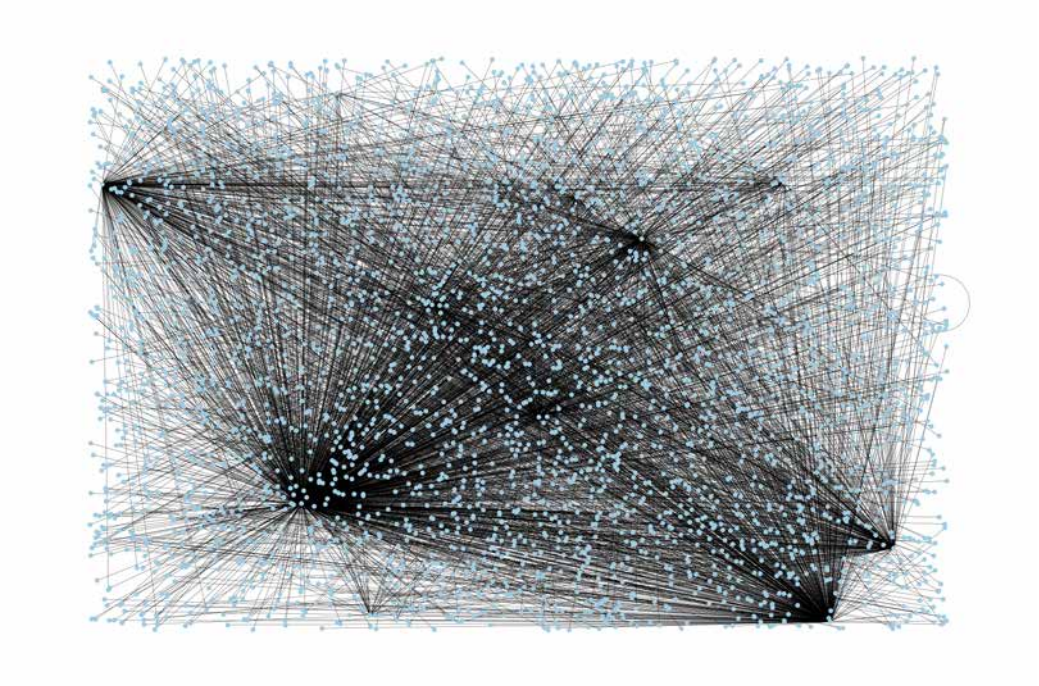

Abbildung 14: Visualisierung eines Graphen aus ca. 6.000 Knoten, eigene Darstellung

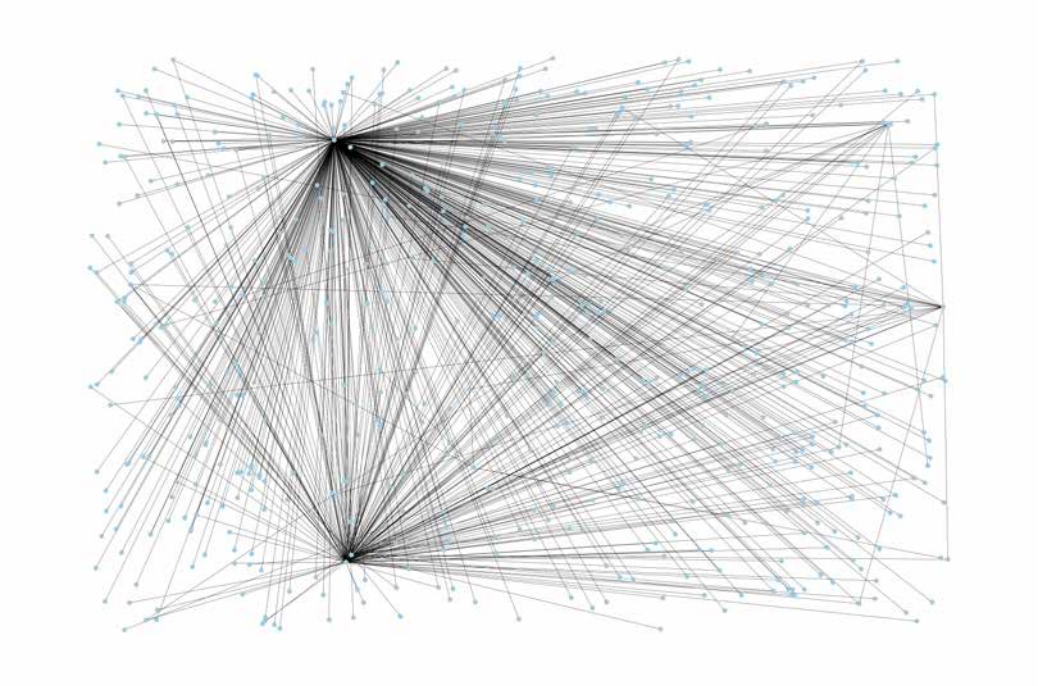

Abbildung 15: Visualisierung eines Graphen aus ca. 100 Knoten, eigene Darstellung

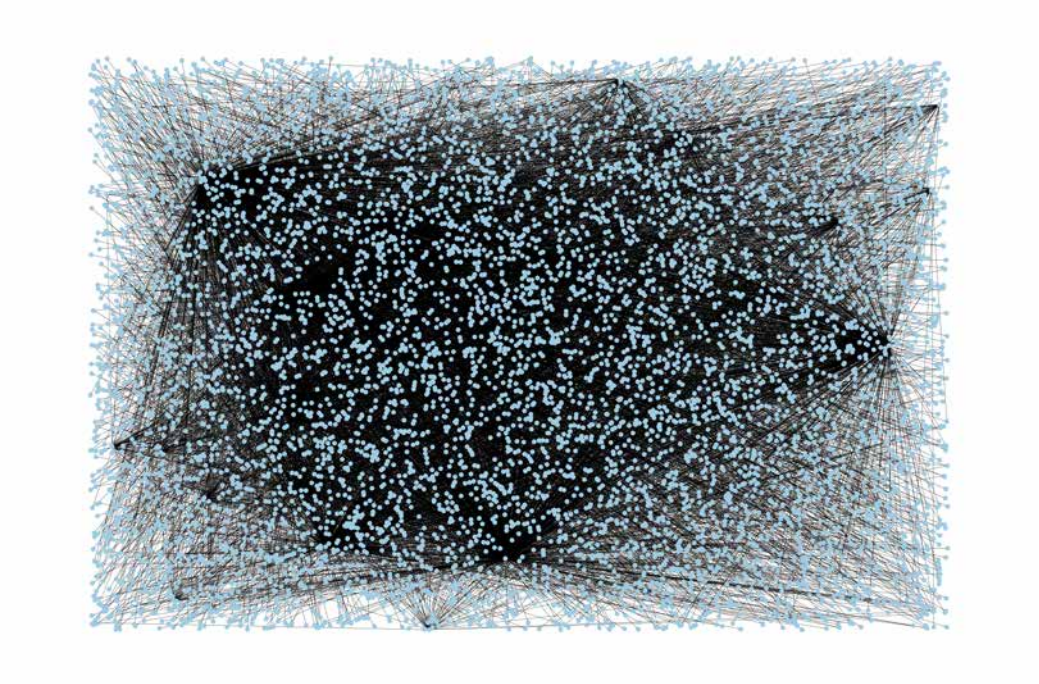

Abbildung 16: Visualisierung eines Graphen aus ca. 14.000 Knoten, eigene Darstellung

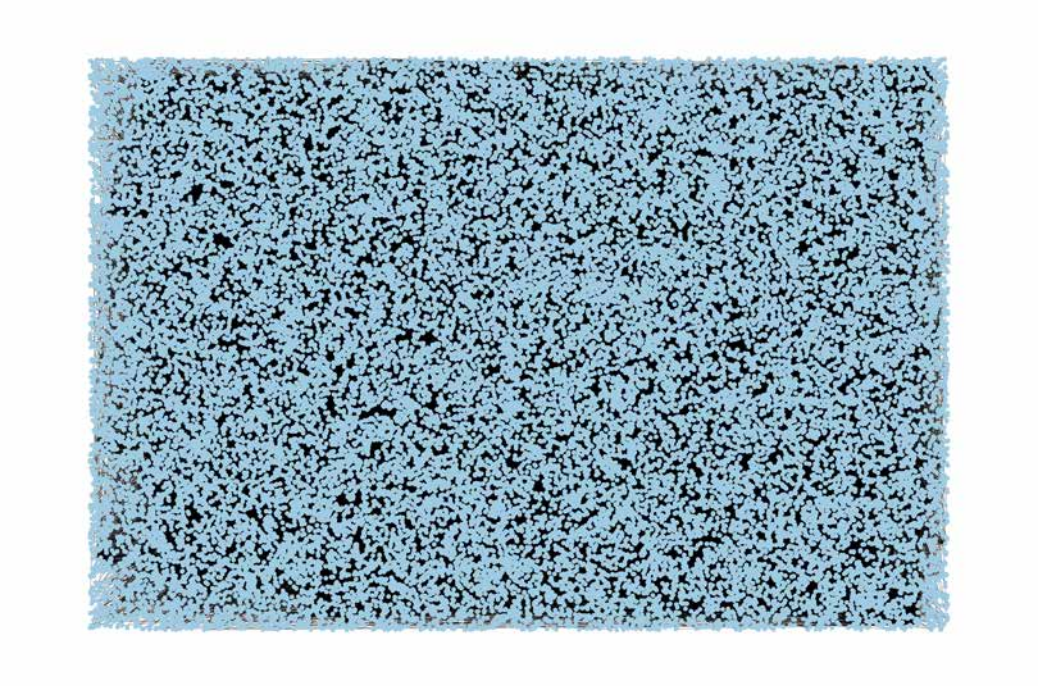

Abbildung 17: Visualisierung eines Graphen aus ca. 36.000 Knoten, eigene Darstellung

Literatur

[Bro+00] Andrei Broder u. a. „Graph structure in the web“. In: *Computer Networks* 33 (2000), S. 309–320.

[Bur21] Bo Burnham. *Welcome to the Internet*. 2021.

[Cas04] Carlos Castillo. „Effective Web Crawling“. Diss. 2004.

[Die06] Reinhard Diestel. *Graphentheorie. Elektronische Ausgabe 2006*. Springer-Verlag, 2006.

[gen21] genius. *Welcome to the Internet, Genius Annotation*. 2021. URL: `https://genius.com/23155625` (besucht am 06. 11. 2021).

[KN09] Sven Oliver Krumke und Hartmut Noltemeier. *Graphentheoretische Konzepte und Algorithmen*. Vieweg, 2009.

[Naj09] Marc Najork. *Web Crawler Architecture*. 2009.

[Swe19] Al Sweigart. *Automate the Boring Stuff with Python, 2nd Edition*. No Starch Press, Incorporated, 2019. Kap. 12.

[Tur10] Volker Turau. *Algorithmische Graphentheorie*. Oldenbourg, 2010.

[wel16] welt. *Zahlen und Fakten zu Wikipedia*. 2016. URL: `https://www.welt.de/newsticker/dpa_nt/infoline_nt/computer_nt/article150992516/Zahlen-und-Fakten-zu-Wikipedia.html` (besucht am 06. 11. 2021).

[Wik21a] Wikipedia. *Statistik*. 2021. URL: `https://de.wikipedia.org/wiki/Spezial:Statistik` (besucht am 06. 11. 2021).

[Wik21b] Wikipedia. *Wikipedia:Size of Wikipedia*. 2021. URL: `https://en.wikipedia.org/wiki/Wikipedia:Size_of_Wikipedia#Number_of_pages` (besucht am 06. 11. 2021).

[YMH02] X. Yuan, M.H. MacGregor und J. Harms. „An Efficient Scheme to Remove Crawler Traffic from the Internet“. In: *Computer Communications and Networks* (2002).

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.
Insbesondere versichere ich, dass ich alle wörtlichen und sinngemäßen Übernahmen aus den Werken als solche kenntlich gemacht habe.

München, den 9. November 2021

Unterschrift