Institut für Formale Methoden der Informatik
Abteilung Formale Konzepte
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3410

Kürzeste Wege im Wikipedia-Linkgraph

Ferdi Kara

Studiengang:	Informatik
Prüfer:	Prof. Dr. Stefan Funke
Betreuer:	Dipl.-Inf. Jochen Eisner Prof. Dr. Stefan Funke Nikola Milosavljevic , PhD Dipl.-Inf. Martin Seybold
begonnen am:	23. Oktober 2012
beendet am:	24. April 2013
CR-Klassifikation:	G.2.2, I.2.8

Kurzfassung

Diese Arbeit beschäftigt sich mit unterschiedlichen Beschleunigungstechniken zur Suche kürzester Pfade in einem Graph. Im Gegensatz zu klassischen Weganfragen wird jedoch kein geographischer Graph als Datenquelle genutzt, sondern der manuell extrahierte Wikipedia-Linkgraph.
Um eine Vergleichsgrundlage für Beschleunigungsalgorithmen zu erhalten, wird eine Auswertung der Breitensuche als Basis geschaffen.
Zur optimalen Auswahl eines Beschleunigungsalgorithmus ist es unabdingbar, ein grundlegendes Verständnis über die Struktur des Graphen zu erhalten. In Folge dieser Untersuchung und einer Vorstellung unterschiedlicher Beschleunigungsalgorithmen wird das Transitknotenkonzept, welches in der Arbeit von Bast u.a. [BFM+07] vorgestellt wurde, auf den Wikipedia-Linkgraph angewandt. Um das Konzept auf einen nicht geographischen Graph anwenden zu können, wird nach der Arbeit von Eisner/Funke [EF12] die Suche nach einer passenden Transitknotenmenge als Hitting-Set-Problem formuliert.
Die Qualität der ausgewählten Transitknoten wird mit unterschiedlichen Konstruktionen zur Transitknotenbestimmung verglichen und die verschiedenen Lösungen werden anhand der vorhergehenden Untersuchung der Graphstruktur erklärt.
Schlussendlich wird gezeigt, warum die verschiedenen Konstruktionen der Transitknotenmenge schlechte Ergebnisse liefern, wodurch das Transitknotenkonzept angewandt auf den Wikipedia-Linkgraph fehlschlägt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Algorithmen

1. Einleitung

Die traditionellen Problemstellungen des "Kürzesten Pfads" behandeln die Problematik der Wegfindung auf Basis von Straßengraphen. Jedoch sind die Anwendungsmöglichkeiten damit noch lange nicht ausgeschöpft, beispielsweise sei hier das Routing in Netzwerken oder auch DNA Sequenz Analysen angeführt.
Mit dem vermehrten Aufkommen von sozialen Netzwerken besteht eine weitere Nutzungsmöglichkeit der kürzeste Pfad Algorithmen im Auffinden von gemeinsamen Freunden. Je kürzer eine Person über gemeinsame Freunde mit einer anderen Person verbunden ist, desto "näher" sind diese befreundet. Wird dieser Gedanke fortgeführt, so kann er auch auf den Wikipedia-Linkgraph angewendet werden, in der eine kurze Entfernung als Indikator für den kontextuellen Zusammenhang zwischen verschiedenen Artikeln verwendet werden kann.
Der Standardalgorithmus für die Suche nach kürzesten Wegen in einem ungewichteten Graph ist die Breitensuche. Durch die Erforschung neuer Beschleunigungsalgorithmen besteht die Möglichkeit, die naive Breitensuche zu beschleunigen.
Darauf aufbauend wird diese Arbeit den Wikipedia-Linkgraph als Datenquelle nutzen und nach einer einleitenden Untersuchung diverser Beschleunigungstechniken den Beschleunigungsalgorithmus um Transitknoten implementieren. Damit diese Entscheidung präzise gefällt werden kann, muss zuvor jedoch ein tiefergehendes Verständnis über den Wikipedia Graph und dessen Eigenschaften und Besonderheiten geschaffen werden, weshalb eine Analyse des Graphen unabdingbar ist.
Nach dem die genannten Strukturen untersucht und die Algorithmen vorgestellt worden sind, werden die Ergebnisse dargestellt und bewertet.
Anschließend werden Anknüpfungspunkte vorgestellt.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Der Wikipedia Graph: Hier wird zunächst die Datengrundlage dieser Arbeit und die einhergehende Datenextraktion beschrieben.

Kapitel 3 – Analyse des Wikipedia-Linkgraph: Eine umfassende Untersuchung der Struktur und der Eigenschaften des Wikipedia-Linkgraphen.

Kapitel 4 – Breitensuche: Implementierung und Benchmark der naiven Breitensuche.

Kapitel 5 – Beschleunigungstechniken: Vorstellung diverser Beschleunigungsschemata.

Kapitel 6 – Transitknotenkonzept angewandt auf den Wikipedia-Linkgraph: Implementierung und Anwendung des Transitnotenkonzeptes.

Kapitel 7 – Zusammenfassung und Ausblick fasst die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte vor.

2. Der Wikipedia Graph

Die Wikipedia ist ein freies und in mehreren Sprachen erhältliches Online-Lexikon, welches offiziell ihren Anfang am 10. Januar 2001 als Teil der Nupedia nahm. Kurze Zeit später, genauer am 15. Januar 2001, wurde die Wikipedia aus der Nupedia getrennt und erreichte damit die eigenständige Webpräsenz, die sie auch heute noch darstellt. Sie wird von der gemeinnützigen Organisation Wikimedia Foundation Inc. mit Sitz in San Francisco, Kalifornien mit Kat Walsh als Vorsitzende, betrieben [Wik13b].
Damit das Ziel einer frei lizenzierten und qualitativ hochwertigen Enzyklopädie erreicht werden kann, hat jeder Internetnutzer neben der Lesemöglichkeit auch die Option, eigene Artikel zu verfassen und auf diese Weise direkt am Entstehungs- bzw. Weiterentwicklungsprozess der Wikipedia mitzuwirken. Durch dieses Konzept ist sie zu einer der meistbesuchten Seiten im Internet (*Alexa Rank 6*)[1] geworden.
Damit die stark wachsende Enzyklopädie gut und mit nur wenig IT-Vorwissen verwaltet werden kann, wurde die frei verfügbare MediaWiki (unter der GPL-Lizenz) entwickelt. Die auf Skalierbarkeit und Funktionalität hin entwickelte Software wurde mit der Skriptsprache PHP realisiert und hat eine Backendanbindung an eine MySQL Datenbank. Diese Software ermöglicht es Nutzern, im Wikitext Format an der Wikipedia zu arbeiten, sodass hier keine Kenntnisse zu xHTML oder CSS vorausgesetzt werden. Eine weitere Besonderheit ist das Versionssystem, welches jeden Artikel in jeder ursprünglichen Form in der Historie speichert, sodass ein Rückfallen oder Vergleich mit älteren Artikeln zu jeder Zeit ermöglicht wird. MediaWiki erlaubt daneben auch Bild- und Multimediadateien zu verwalten.
Damit diese Menge an Informationen entsprechend effizient von MediaWiki verwaltet werden kann, hat die Software derzeit 26 Namensräume definiert. Tabelle 2.1 und Tabelle 2.2 zeigen die verschiedenen Namensräume mit einer entsprechenden Kurzbeschreibung auf. Zusammenfassend kann die Wikipedia in 12 Basisnamensräume mit je einem zusätzlichem Diskussionsnamensraum und 2 virtuellen Namensräumen aufgeteilt werden.

[1] `http://www.alexa.com/`

Namensräume		
Nummer	Präfix	Bedeutung
0	Main	Der Artikelnamensraum, in dem die aktuell verfügbaren Artikel der Enzyklopädie präsent sind
1	Talk	Diskussionsseiten rund um Artikel im Namensraum 0
2	User	Persönliche Benutzerseiten
3	User talk	Diskussionsseiten rund um Benutzerseiten aber auch sonstige Diskussionen
4	Wikipedia	Interne Seiten zur Wikipedia wie allgemeine Informationen und Regeln
5	Wikipedia talk	Diskussionsseiten rund um Namensraum 4
6	File	Informationen zu Medien-Dateien und dem Link zur Medien-Datei selbst
7	File talk	Diskussionsseiten rund um Namensraum 6
8	MediaWiki	Die Texte der MediaWiki
9	MediaWiki talk	Diskussionsseiten rund um Namensraum 8
10	Template	Vorlagen für das einfachere Erstellen von häufig genutzten Elementen
11	Template talk	Diskussionsseiten rund um Namensraum 10
13	Help	Hilfeartikel rund um die Wikipedia und die Software
13	Help talk	Diskussionsseiten rund um Namensraum 12
14	Category	Listet die verfügbaren Kategorien auf und ermöglicht eine Auflistung von Artikeln nach gewählten Kategorien
15	Category talk	Diskussionsseiten rund um Namensraum 14
100	Portal	Portalseiten
101	Portal talk	Diskussionsseiten rund um Namensraum 100
108	Book	Artikel, die unabhängig von der Wikipedia angeboten werden (als optimierter elektronischer Artikel oder auch als bestellbares ausgedrucktes Dokument)
109	Book talk	Diskussionsseiten rund um Namensraum 108
446	Education Program	Artikel zu Bildungsprogrammen
447	Education Program talk	Diskussionsseiten rund um Namensraum 446

Tabelle 2.1.: Wikipedia Namensräume

Fortsetzung Namensräume		
Nummer	Präfix	Bedeutung
710	TimedText	Synchronisation eines Textmediums mit anderen Medien (beispielsweise als Untertitel in einem Video oder Teleprompter-Anwendungen)
711	TimedText talk	Diskussionsseiten rund um Namensraum 710
Virtuelle Namensräume		
-1	Special	Automatisch generierte Spezialseiten (Letzte Änderungen, verwaiste Seiten, alle Artikel Auflistung, ...)
-2	Media	Direkter Verweis zu einer Medien-Datei (ohne Umweg über die Beschreibungsseite)

Tabelle 2.2.: Fortsetzung: Wikipedia Namensräume

Die schon erwähnte freie Verfügbarkeit ermöglicht es ihren Nutzern, den Wikipediainhalt nicht nur online zu betrachten, sondern bietet darüber hinaus vielfältige Downloadmöglichkeiten ihrer Datenbank an. Der textuelle Inhalt wird hierbei unter der Creative Commons Attribution-ShareAlike 3.0 License (CC-BY-SA) und unter der GNU Free Documentation License (GFDL) veröffentlicht. Bilder und andere Dateien, die in den Namensraum Media fallen, dürfen aus einer Anzahl von Lizenzen wählen, wobei immer eine kommerzielle Wiederverwendung und eine Nutzung für daraus abgeleitete Werke möglich sein muss. Darunter fallen unter Anderen die GNU Free Documentation License, Creative Commons, Free Art, Attribution und Public domain license. Durch den stark wachsenden Umfang der Wikipedia gilt hierbei zu beachten, dass das Betriebssystem, auf welchem die Daten bearbeitet werden sollen, ein entsprechendes Dateisystem, welches große Dateien unterstützt, anbietet.

Die angebotenen Downloads unterteilen sich in:

- Datenbank Sicherungskopien
- Liste der Spiegelseiten der Wikipedia
- DVD-Versionen
- Statische HTML Version (aktuell nicht mehr fortgesetzt)
- Sicherungskopien von Wikimedia Artikeln, die so nicht mehr in der Hauptdatenbank gesichert sind (im Speziellen der Artikel um den 11.September)
- Sonstige Dateien (wie Statistiken der Seitenaufrufe, Archive aus Bildern, Umfragedaten, und weitere)

- Speziell aufbereitete Datensektkollektionen (interessant für Forscher und Entwickler)

Eine genauere Auflistung aller möglichen Datenbank Sicherungskopien ist unter Tabelle 2.3 und Tabelle 2.4 zu finden.

Datenbanksicherungen		
Name	Format (Archiv)	Inhalt
pages-articles	xml (bz2)	Enthält die aktuelle Version aller Artikelseiten, Vorlagen, Medienbeschreibungen und die primären Meta-Seiten
pages-meta-current	xml (bz2)	Enthält die aktuelle Version aller Artikelseiten inklusive Diskussions- und Benutzerseiten
pages-meta-history	xml (7z)	Enthält die aktuelle und jede revisionierte Version aller Artikelseiten inklusive Diskussions- und Benutzerseiten
flaggedrevs	sql (gz)	Enthält markierte Revisionen mit Informationen zu den Markierungen (Wer, Wann, Warum, Art, Qualität)
fallgedpages	sql (gz)	Enthält markierte Artikel mit Informationen zu den Markierungen (RevisionsID, letzte Änderung markiert?, Wartedauer anstehender Änderungen)
pages logging	xml (gz)	Alle Log Ereignisse
stub-meta	xml (gz)	Enthält ausschließlich Revisions Metadateien mit einer Unterteilung in *history*, *current* und *articles*
abstract	xml	Enthält nur den Artikel Abstract
all-titels-in-nso	txt (gz)	Alle Artikeltitel
iwlinks	sql (gz)	Interwiki Verweisverlauf
redirect	sql (gz)	Liste der Weiterleitungen
protected_titles	sql (gz)	Liste von Titeln, die geschützt sind, sodass keine Artikel darunter erstellt werden können
page_props	sql (gz)	Name/Wert Paare für Seiten
page_restrictions	sql (gz)	Auflistung aller Beschränkungen für jede Seite
page	sql (gz)	Grundlegende Informationen für jede Seite (ID, Titel, Beschränkungen, ...)
category	sql (gz)	Kategorie Informationen

Tabelle 2.3.: Auflistung der Datenbanksicherungen

Datenbanksicherungen		
Name	Format (Archiv)	Inhalt
user-groups	sql (gz)	Auflistung der vorhandenen Nutzergruppen
interwiki	sql (gz)	Menge aus vordefinierten Interwiki Präfixen
langlinks	sql (gz)	Ausgehende Verweise auf die selben Artikel in anderen Sprachen
externallinks	sql (gz)	Alle externen Verweise
templatelinks	sql (gz)	Alle Vorlagen Verweise
imagelinks	sql (gz)	Informationen zur Nutzung von Mediendateien auf Wikiseiten
categorylinks	sql (gz)	Liste aller Zugehörigkeiten zu Kategorien für Wikiseiten
pagelinks	sql (gz)	Interne Wiki Verweise
oldimage	sql (gz)	Metadaten über veraltete Versionen von Mediendaten
image	sql (gz)	Metadaten über aktuelle Versionen von Mediendaten
site_stats	sql (gz)	Diverse Statistiken (z. B. Seitenaufrufe)

Tabelle 2.4.: Fortsetzung: Auflistung der Datenbanksicherungen

Die durchschnittliche Dauer einer vollen Sicherung der englischen Wikipedia dauert 8-9 Tage (Quelle: [Wik13a]). Zum Zeitpunkt dieser Arbeit wird zu jedem Monatsanfang eine Sicherung gestartet und die Ergebnisse der diversen Sicherungen, wie in Tabelle 2.3 und Tabelle 2.4 beschrieben, direkt zum Herunterladen zur Verfügung gestellt. Durch die immense Größe der Daten werden entsprechende Teilarchive angeboten.

2.1. Extraktion

Um eine geeignete Datenbasis für diese Arbeit zu schaffen, ist die trivialste Lösung das Benötigte aus der Datenbanksicherung *pages-articles* zu extrahieren. Da diese jedoch unnötig viele Daten enthält, ist der effizientere Weg sich die Daten aus den unterschiedlichen Datenbanksicherungen selbst zusammenzubauen.

Hierbei bietet die Datenbanksicherung *page* eine Sammlung aus SQL Befehlen an, die eine SQL Tabelle erstellt, in der die grundlegenden Informationen eingefügt werden. Das Format sieht folgendermaßen aus:

(*page_id*, *page_namespace*, *page_title*, *page_restrictions*, *page_counter*, *page_is_redirect*, *page_is_new*, *page_random*, *page_touched*, *page_latest*, *page_len*)

Da jede Zeile mehrere Anweisungen enthält, werden die einzelnen SQL Befehle (in der SQL Syntax getrennt durch ein Komma) mit einem regulären Ausdruck extrahiert und es wird daraufhin überprüft, ob der aktuell betrachtete Titel im Namensraum 0 liegt (d. h. es werden ausschließlich Artikel betrachtet). Falls dies zutrifft, wird der Titelstring in eine neue Datei *keys* geschrieben, wobei jede Zeilennummer entsprechend die ID darstellt.
Durch die Tatsache, dass nur ein Teil der Titel extrahiert wird, entstehen bei der Nummerierung der Titel IDs große Lücken, die mit einer Sortierung der extrahierten Titel und Speicherung in eine neue Datei *keys_sorted* gelöst werden. Damit entsteht die benötigte lückenlose Nummerierung der Titel IDs. Um eine möglichst effiziente Sortierung zu gewährleisten, wird ein Mergesort Algorithmus genutzt, der die Elemente in $\mathcal{O}(n\ log(n))$ sortiert. Sortieralgorithmen, die zwar keinen zusätzlichen Speicherplatz für die Sortierung benötigen, jedoch in $\mathcal{O}(n^2)$ liegen, haben sich bei dieser Datenmenge als unpraktikabel erwiesen.

Der nächste Schritt ist die Extraktion der Verweise. Hierbei wird die Sicherung *pageslinks* benötigt, die ebenfalls ähnlich der Sicherung *page* eine Sammlung aus SQL-Befehlen darstellt, die eine entsprechende Tabelle in einer Datenbank erstellt, auf die jeder interne Wiki Verweis eingefügt wird. Das Format dieser Einfügungen sieht wie folgt aus:

$$(page\_id,\ page\_namespace,\ page\_title)$$

Zu Beachten ist hierbei, dass der Startpunkt per ID und der Zielpunkt als Titelstring und Namensraum angegeben ist. Ähnlich wie bei der Titelstringextraktion aus *page* werden hier ebenfalls per regulärem Ausdruck die einzelnen Anweisungen in den Speicher geladen und unter den folgenden zwei Bedingungen in eine neue Datei *pages_links_temp* geschrieben: Der Zieltitel muss im Namensraum 0 liegen und der Starttitel muss einer ID in der Datei *keys* zugewiesen sein. Falls dies zutrifft, werden die Verweise als Strings getrennt durch ein Leerzeichen in eine temporäre Datei geschrieben. Das folgende Format wird dabei genutzt:

$$Titel\_des\_Ausgangsartikels\ Zieltitel\_1\ Zieltitel\_2\ Zieltitel\_3\ ...\backslash n$$

Dieser Umweg über eine temporäre Datei *pages_links_temp* ist deshalb nötig, weil nicht alle Verweise (durch die Neusortierung der Titelstrings in der Datei *keys_sorted* herrscht hier keine Ordnung vor) in diesem Format im Speicher gehalten werden kann und das Schreiben in unterschiedliche Zeilen in Java (ohne genauere Informationen über die Datei) ein Durchlaufen der vorherigen Zeilen voraussetzt (im schlechtesten Fall liegt das mit n Artikel und m Verweisen in $\mathcal{O}(m^2)$).
Der abschließende Schritt beinhaltet das Parsen der Titel in der Datei *pages_links_temp* auf die neuen IDs, welche in der Datei *keys_sorted* hinterlegt sind und so in den Hauptspeicher per Hashtabelle geladen werden können. Dies kann jedoch nicht in einem Schritt gemacht werden, da die Strings ungeordnet sind, weshalb hier immer nur ein stückchenweises Einlesen und Parsen erfolgt. Ab einer Grenze wird der schon auf die neuen IDs geparste Bereich des Hauptspeichers auf die Festplatte (als *pages_links_out_temp_I*) entleert und es wird bei dem noch nicht geparsten Bereich der Datei *pages_links_temp* fortgesetzt. Das erneute Niederschreiben erfolgt nun nicht mehr nur in eine neue Datei (*pages_links_out_temp_I+1*), sondern es werden die Zeilen, in die nichts geschrieben wird, von der vorherigen Datei (*pages_links_out_temp_I*) kopiert.

So entsteht nach und nach die Datei *pages_links_out*, die alle ausgehenden Verweise in alphabetischer Ordnung enthält. Diese werden im folgenden Format dargestellt, wobei die Zeilennummer die ID des Startknotens wiedergibt:

$ZielID\_1\ ZielID\_2\ ZielID\_3\ ...\backslash n$

2.2. Effiziente Darstellung

In vielen einleitenden Lehrbüchern wird eine Graphdarstellung in Java klassisch objektorientiert realisiert. Das heißt, dass eine Klasse Graph erstellt wird, in der Kanten mit Hilfe von Knotenobjekten realisiert werden. Dies ist jedoch für große Graphen nicht anwendbar. Um eine möglichst effiziente Darstellung des Graphen zu gewährleisten, muss erst ein Verständnis dafür entwickelt werden, wie Java Speicherplatz verwaltet. Der primitive Datentyp int in einer 32bit (64bit) Java Virtual Machine belegt beispielsweise 32bit (64bit) Speicher, wohingegen das Objekt Integer neben dem 32bit (64bit) Payload noch zusätzliche 96bit (192bit) für die Verwaltung und 32bit (64bit) für jede weitere Referenz darauf aufschlägt (siehe [Roso8] für mehr Informationen). Mit Hilfe dieser kurzen Erläuterung ist es klar, wieso primitive Datentypen genutzt werden sollten falls die ganze Struktur im Hauptspeicher verwalten werden soll. Die intuitiv triviale Variante würde einfach 2 int Arrays der Größe m initialisieren. In das erste Array wird der Startknoten und in das zweite Array der Zielknoten eingetragen, womit eine Speicherplatzbelegung in der Größe von $\mathcal{O}(2m)$ erreicht wird. Da auf die Struktur des Wikipedia Graphen erst in Kapitel 3 genauer eingegangen wird, sei hier mit der folgenden Information etwas voraus gegriffen. Das Verhältnis der Anzahl der Kanten zur Anzahl der Knoten beträgt 29:1, wodurch bei der trivialen Lösung das Knotenarray eine geringere Datenmenge enthält. Um eine effizientere Darstellung zu ermöglichen, wird ein Offset Array der Größe n und ein Datenarray der Größe m initialisiert.

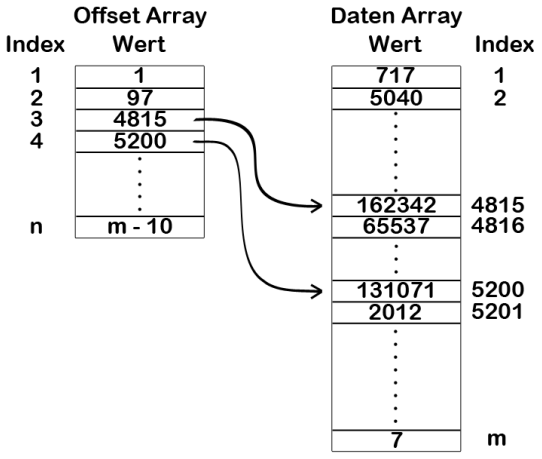

Abbildung 2.1.: Offset Array Darstellung

Wie in Abbildung 2.1 gezeigt wird, steht im Offset Array als Index die KnotenID und als Wert die Position bzw. der Index der Daten, die im Daten Array gespeichert sind. Der Pseudocode 2.1 erläutert die Zugriffsmethode genauer.

Algorithmus 2.1 Zugriff auf Kanten von Knoten

```
int laenge
int[] ergArray
if knoten_id ≠ groesste_auftretende_ID then
    laenge = offsetArray[knoten_id + 1] − offsetArray[knoten_id]
else
    laenge = anzahl_Kanten − offsetArray[knoten_id]
end if
for i = 0 → laenge do
    ergArray[i] = datenArray[offsetArray[knoten_id] + i]
    i ← i + 1
end for
return ergebnisArray
```

So stehen in der Abbildung 2.1 die Kantenwerte für den Knoten mit der ID 3 an Stelle 4815 bis (exklusive) 5200 des Daten Arrays.
Damit wird nur Speicherplatz in der Größenordnung $\mathcal{O}(n+m)$ benötigt.

2.3. Extraktion der eingehenden Verweise

Für die spätere Bearbeitung sollten auch die eingehenden Kanten direkt verfügbar sein.

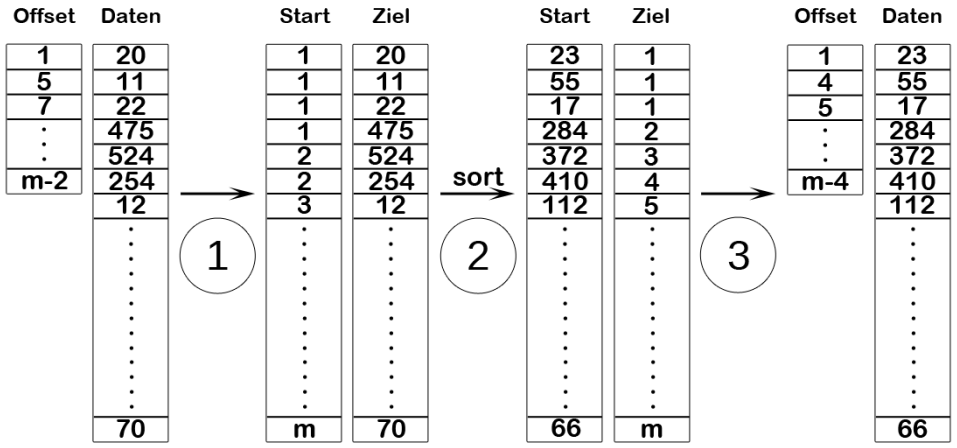

Abbildung 2.2.: Konvertierung zu eingehenden Kanten

Um aus der Offset Datenstruktur der ausgehenden Kanten eine entsprechende Datenstruktur für eingehende Kanten zu erstellen, sind folgende Schritte nötig (vergleiche Abbildung 2.2):

1. Schreibe das Offset Array in das Start Array aus und kopiere das Daten Array in das Ziel Array.
2. Sortiere die verknüpften Arrays ausgehend von der Ziel Array Ordnung.
3. Erstelle anhand von Start und Ziel Array eine Offset Struktur (dabei wird das Ziel Array verworfen), wobei nun das Start Array als Daten Array genutzt wird.

2.4. Serialisierung

Die bis dato beschriebenen Verfahren schreiben alle Daten als Textdateien ins Dateisystem. Das erneute Einlesen der gesamten Graphstruktur kann mitunter eine gewisse Zeit in Anspruch nehmen.
Um diese Prozess zu beschleunigen, wird eine zuvor eingelesene Graphstruktur in einem binären Format gesichert. Dies wird mit der Java Object Serialization (JOS) realisiert. Hierbei muss das Graphobjekt die Schnittstelle *java.io.Serializable* implementieren und die Attribute sollten Basistypen oder selbst serialisierbare Objekte sein. Der erzielte Geschwindigkeitsgewinn kann in Tabelle 2.5 betrachtet werden.

Einleseoptionen	Parsen auf HDD	binäres Format
I + O	107	5
I + O + HM + TS	122	62

Tabelle 2.5.: Einlesezeiten (in Sekunden)
I = eingehende Kanten, **O** = ausgehende Kanten, **HM** = Einlesen der Titelstrings in eine Hashtabelle, **TS** = Einlesen der Titelstrings in ein String Array (Systeminformationen)

3. Analyse des Wikipedia-Linkgraph

Nach der Extraktion der benötigten Daten liegen nun alle Daten in einem für die Analyse des Graphen geeigneten Format vor.
Durch die allgemeine Aufbaustruktur von Webseiten kann jedoch schon vor der eigentlichen Untersuchung angenommen werden, dass der Graph nicht all zu sehr in die Tiefe gehen sollte (keine langen Wege) jedoch dafür sehr stark in die Breite geht (viele kurze Wege).
Um hierbei einen Vergleich zu einem klassischen Straßengraphen ziehen zu können, wird diese Arbeit den Graph des US Straßennetzes als Referenz nehmen, welcher aus 24 Millionen Knoten und 58 Millionen Kanten besteht. Das System, auf welchem eventuelle Benchmarks und Speicherplatzverbrauch von Algorithmen gemessen werden, kann unter Tabelle A.1 aufgerufen werden.

3.1. Allgemeine Informationen

Diese Arbeit wird im Detail die Sicherung der englischen Wikipedia (kurz EN) vom 02. August 2012 und als zusätzliche Vergleichsgrundlage die Simple English Wikipedia (kurz SIM) vom 03. Februar 2013 verwenden, welcher in einer vereinfachten Form der englischen Sprache geschrieben ist. Des weiteren werden im Anhang die Ergebnisse für die deutsche Wikipedia vom 28. Januar 2013 und für die französische Wikipedia vom 25. Januar 2013 ergänzt.
Nach der Extraktion der Artikel im Namensraum 0 (siehe Abschnitt 2.1) verkleinert sich die Graphstruktur, die ursprünglich für den EN Graph aus 40 Millionen Knoten bestand, auf die Werte in Tabelle 3.1.

Graph Details	EN	SIM
Knoten	9.591.525	126.201
Kanten	267.532.460	3.941.294
Knoten ohne eingehende Kanten	3.461.186	24.123
Knoten ohne ausgehende Kanten	5.455	672
∅ Kanten pro Knoten	27,89	31,23

Tabelle 3.1.: Allgemeine Informationen zum Linkgraph

Mit nur 9,6 Millionen Artikeln bzw. Knoten hat der Linkgraph im Vergleich zum Straßengraphen weniger als die Hälfte der Knoten, jedoch ist die Kantenanzahl um ein Vielfaches größer als in einem Straßengraphen. Diese Eigenschaft bestätigt die anfängliche Vermutung von einem breiten Graphen, denn wo der Straßengraph im Durchschnitt nur 2,42 Kanten pro Knoten aufweist, ist der Linkgraph mit 27,89 Knoten (bzw. 31,23 Knoten für SIM) sehr viel stärker aufgespannt.
Eine weitere interessante Eigenschaft ist die hohe Anzahl von Knoten, auf die keine Kante eingeht, also auf die nicht verlinkt wird. Dies ist teilweise damit zu begründen, dass Artikel durch verschiedene Titelstrings (Abkürzungen, unterschiedliche Schreibweisen, etc.) repräsentiert sind (für den Nutzer ist dies durch *Weitergeleitet von ...* ersichtlich), sodass der Hauptartikel mit dem eigentlichen Inhalt von vielen Nebenartikeln, die im Grunde nur aus einer Weiterleitung bestehen, referenziert werden, auf die selbst natürlich nirgends verwiesen wird.

Grad	Für ausgehende Kanten		Für eingehende Kanten	
	Bereich	Gesamtanzahl	Bereich	Gesamtanzahl
0	5.455	5.455 (0,06%)	3.461.186	3.461.186 (36,09%)
1	5.524.890	5.524.890 (57,6%)	1.221.769	1.221.769 (12,74%)
2 - 20	59.807 - 109.662	1.729.726 (18,03%)	39.861 - 714.221	3.145.814 (32,8%)
21 - 40	26.925 - 55.679	736.915 (7,68%)	19.723 - 27.046	542.431 (5,66%)
41 - 60	15.310 - 25.264	400.441 (4,17%)	11.654 - 18.781	301.903 (3,15%)
61 - 80	11.304 - 15.048	270.675 (2,82%)	8.153 - 11.700	198.848 (2,07%)
81 - 100	8.152 - 12.079	196.176 (2,05%)	5.828 - 8.127	139.202 (1,45%)
101 - 200	2.124 - 8.150	440.774 (4,6%)	1.638 - 5.702	319.233 (3,33%)
201 - 300	850 - 2.332	148.599 (1,55%)	601 - 1.851	115.212 (1,2%)
301 - 400	278 - 1.084	61.225 (0,64%)	235 - 882	49.282 (0,51%)
401 - 500	136 - 961	29.675 (0,31%)	137 - 615	29.044 (0,3%)
501 - 1000	5 - 763	42.577 (0,44%)	14 - 575	50.977 (0,53%)
> 1000	0 - 333	4.397 (0,05%)	0 - 917	16.624 (0,17%)

Tabelle 3.2.: Untersuchung der Knotenvalenz (EN)

Tabelle 3.2 zeigt ein genaueres Bild von der Knotenvalenz für ausgehende Kanten. So gibt es 5,5 Millionen Knoten, die einen Ausgangsgrad von 1 haben, welches über die Hälfte aller Knoten darstellt. Des Weiteren beträgt beispielsweise die Anzahl der Knoten mit einem Ausgangsgrad, welcher zwischen 2 bis 20 liegt, 1.729.726 Knoten. Die individuellen Werte für jeden Knoten würde den Rahmen sprengen, weshalb hier nur ein Bereich eingetragen ist, in denen die Anzahl der Knoten mit dem jeweiligen Knotengrad liegen. Für das erwähnte Beispiel bedeutet dies, dass Knoten mit dem Ausgangsgrad 2 bis 20 ein individuelles Vorkommen im Bereich von 59.807 bis 109.662 haben; beispielsweise gibt es 90.529 unterschiedliche Knoten mit dem Ausgangsgrad 2.
Eine entsprechende Untersuchung für eingehende Kanten zeigt, dass vor allem die Anzahl

der Knoten ohne eingehende Kanten mit 36,09% an der Gesamtknotenanzahl sehr hoch ist. Darüber hinaus ist der Anteil der Knoten mit einstelligem eingehendem Knotengrad, welcher für Knotengrad 10 immer noch bei über 100.000 liegt, weitaus höher als für die entsprechenden Knoten mit ausgehenden Kanten. Schließlich stellt sich der Schlussbereich in dieser Untersuchung ebenfalls unterschiedlich dar. So gibt es für Knoten mit Grad größer 1.000 knapp 4 mal mehr Knoten. Wo bei ausgehenden Kanten die Grenze bei 8.103 Kanten pro Knoten liegt, erstreckt sich dieser Wert für eingehende Kanten auf 749.058 Kanten (vergleiche Tabelle 3.1, Tabelle 3.4 und Tabelle 3.6). Dies ist trivialerweise über die Struktur eines Artikels zu erklären. So hat ein Artikel selten viele ausgehende Links im Vergleich zu eingehenden Kanten, denn vor allem die allgemeinen Artikel werden häufig von vielen Artikeln heraus verlinkt, wodurch sich die teilweise stark unterschiedliche Knotenvalenz begründet. Mit diesem Hintergrundwissen liegt der Knotengrad für alle Knoten im EN Graph, die eingehende Kanten haben (entspricht etwa 6,1 Millionen Knoten), bei 43,6.

	Für ausgehende Kanten		Für eingehende Kanten	
Grad	Bereich	Gesamtanzahl	Bereich	Gesamtanzahl
0	672	672 (0,53%)	24.123	24.123 (19,11%)
1	37.916	37.916 (30,04%)	22.028	22.028 (17,45%)
2 - 20	1.208 - 6.092	57.027 (45,19%)	661 - 13.538	55.042 (43,61%)
21 - 40	378 - 1.133	13.832 (10,96%)	281 - 786	8.633 (6,84%)
41 - 60	150 - 387	4.970 (3,94%)	104 - 263	3.306 (2,62%)
61 - 80	81 - 206	2.519 (2%)	64 - 260	2.425 (1,92%)
81 - 100	53 - 131	1.457 (1,15%)	43 - 111	1.435 (1,14%)
101 - 200	3 - 182	3.578 (2,84%)	3 - 152	3.560 (2,82%)
201 - 300	0 - 176	1.174 (0,93%)	0 - 539	2.307 (1,83%)
301 - 400	0 - 177	782 (0,62%)	0 - 316	919 (0,73%)
401 - 500	0 - 339	865 (0,69%)	0 - 448	1.384 (1,1%)
501 - 1000	0 - 333	1.406 (1,11%)	0 - 568	940 (0,74%)
> 1000	0 - 1	3 (0%)	0 - 2	99 (0,08%)

Tabelle 3.3.: Untersuchung der Knotenvalenz (SIM)

Für den SIM Graph ergibt sich ein in der Summe leicht verändertes Bild. So ist die Anzahl der Knoten mit Grad 0 und 1 für ausgehende und eingehende Kanten niedriger, dafür kommen jedoch die Knoten mit Grad 2-40 sehr viel häufiger vor. Durch den einfacheren Aufbau des SIM Graph, in der Artikel im Durchschnitt sehr viel kürzer und mit einem simpleren Satzbau aufgestellt sind, wird entsprechend auf weniger Artikel (vor allem auf spezialisierte Artikel) verlinkt. Dies hat das erwähnte relativ häufige Aufkommen der Knoten der Grade 2-40 zur Folge. Die restliche Untersuchung zeigt ein entsprechendes Abnehmen der höheren Knotenvalenz. Das Bemerkenswerte ist jedoch, dass es für einen relativ kleinen SIM Graph immer noch recht viele Knoten mit einem Knotengrad größer 100 gibt. Diese Struktur ist den vielen Listenartikeln geschuldet, die später in Tabelle 3.5 und Tabelle 3.7 genauer

betrachtet werden. Der Knotengrad für den SIM Graph, bei dem ein Knoten mindestens eine eingehende Kante hat (entspricht etwa 100.000 Knoten), liegt bei 38.

Um eine genauere Unterteilung der Strukturierung des EN (bzw. SIM) Graphen zu erhalten, werden in Tabelle 3.4 (bzw. Tabelle 3.5) und Tabelle 3.6 (bzw. Tabelle 3.7) die Artikel nach Anzahl der jeweiligen Kanten sortiert angezeigt.

Titel	Anzahl ausgehender Kanten
Alphabetical_list_of_comuni_of_Italy	8.103
IUCN_Red_List_endangered_animal_species	5.666
List_of_municipalities_of_Brazil	5.516
List_of_years	5.469
Index_of_India-related_articles	5.317
Index_of_philosophy_articles_(I-Q)	5.232
List_of_dialling_codes_in_Germany	4.949
List_of_Ireland-related_topics	4.923
List_of_populated_places_in_Serbia	4.679
List_of_film_director_and_cinematographer_collaborations	4.675

Tabelle 3.4.: Auflistung der Knoten geordnet nach Anzahl ausgehender Kanten (EN)

Titel	Anzahl ausgehender Kanten
List_of_J._League_players	2.039
List_of_Japanese_footballers	1.818
List_of_years	1.430
Marquise,_Pas-de-Calais	907
Communes_of_the_Pas-de-Calais_department	900
2006	899
Deaths_in_2012	862
January_1	856
List_of_municipalities_in_Switzerland	846
Courcelles-sur-Vesles	846

Tabelle 3.5.: Auflistung der Knoten geordnet nach Anzahl ausgehender Kanten (SIM)

Hierbei wird deutlich, dass ausgehende Kanten ihren Ursprung in Index- und Listenartikeln haben (bei Ausweitung der Werte von Tabelle 3.4 auf 100 oder 1000 bleibt diese Eigenschaft bestehen). Mit 8.103 ausgehenden Kanten ist der Artikel *Alphabetical_list_of_comuni_of_Italy* als Artikel mit den meisten ausgehenden Kanten im Graph vertreten. Die Nachfolger

befinden sich in der 5.000 er Region. Die selbe Untersuchung auf dem SIM Graph Tabelle 3.5 bestätigt, dass die vorherige Beobachtung auch für den viel kleineren SIM Graph gültig ist.

Titel	Anzahl eingehender Kanten
Geographic_coordinate_system	749.058
United_States	501.307
International_Standard_Book_Number	332.641
Time_Zone	220.140
Biological_classification	213.090
Music_genre	206.590
Record_label	193.027
Internet_Movie_Database	181.541
United_Kingdom	178.718
France	176.711

Tabelle 3.6.: Auflistung der Knoten geordnet nach Anzahl eingehender Kanten (EN)

Titel	Anzahl eingehender Kanten
United_States	14.902
France	10.671
Media	9.335
Geographic_coordinate_system	7.743
Departments_of_France	6.568
Communes_of_France	6.168
Association_football	5.524
Japan	5.354
Regions_of_France	5.195
City	4.269

Tabelle 3.7.: Auflistung der Knoten geordnet nach Anzahl eingehender Kanten (SIM)

Wenn dazu nun die entsprechenden Datensätze für eingehende Kanten angeschaut werden, kommt eine sehr unterschiedliche Struktur auf. Waren es bei ausgehenden Kanten nur Index- und Listenartikel, so sind es bei eingehenden Kanten eher allgemein gehaltene Artikel (Länder, Alltagsklassifikationen wie ISBN oder UTC, besondere Ereignisse der Geschichte, etc). Auch die Anzahl der Verweise ist hier wesentlich größer. So wird der Artikel *Geographic_coordinate_system* 749.058 mal von anderen Artikeln heraus verlinkt. Der erste Artikel, der eine vergleichbare Anzahl an Verweisen wie der Artikel mit den meisten

ausgehenden Kanten hat, taucht erst an Stelle 943 in der Sortierung für eingehende Kanten auf. Eine entsprechende Beobachtung lässt sich ebenso für den SIM Graph feststellen.
Zusammenfassend zu den beiden Tabellen lässt sich sagen, dass Artikel zwar selber nicht zu vielen Artikeln verlinken, jedoch gehen die vorhandenen Verweise größtenteils auf die selben Artikel, wodurch einige wenige Artikel sehr viele Kanten aufweisen.

3.2. Zusammenhang

Um genauere Informationen zum Graphaufbau zu erhalten, muss die Zusammenhangseigenschaft untersucht werden.
Zur Berechnung des schwachen Zusammenhangs wurde Algorithmus 3.1 verwendet. Die Implementierung kann hierbei mit der Tiefensuche oder mit der Breitensuche erfolgen. Im Grunde läuft dieser Algorithmus alle erreichbaren Knoten von einem Startknoten durch. Falls es noch Knoten gibt, die nicht besucht wurden, wird die Nummer der Komponenten (dient hierbei als Markierung) erhöht und startend von dem unbesuchten Knoten der Algorithmus wiederholt bis alle Knoten besucht wurden (das heißt jedem Knoten wurde eine Komponentennummer zugewiesen).

Algorithmus 3.1 Schwache Zusammenhangskomponenten

```
clearMarks()
clearVisited()
componentNumber=0
for i = 1 → |V| do
    if notVisited(v) then
        componentNumber = componentNumber + 1
        undirectedBfs(v)
    end if
    i ← i + 1
end for
```

Der Algorithmus angewandt auf den Wikipedia-Linkgraph liefert als Ergebnis die Tabelle 3.8.

Komponentengröße	1	2	3	4	5	7	9.587.476
Vorkommen EN	3.730	122	16	1	1	1	1
Komponentengröße	1	2	4	7			125.781
Vorkommen SIM	335	35	2	1			1

Tabelle 3.8.: Auswertung schwacher Zusammenhänge

Der EN Linkgraph hat insgesamt 3.873 Komponenten, wovon einer 9.587.476 Elemente enthält (entspricht 99,96% aller Knoten). Die anderen Komponenten mit den Größen 1,2,3,4,5 und 7 haben damit nur einen verschwindend geringen Anteil an der Gesamtknotenzahl des Graphen. Für den SIM Linkgraph ergibt sich ein ähnliches Bild mit insgesamt 374 Komponenten, wo eine große Komponente nahezu alle Knoten enthält und somit ein verschwindend geringer Anteil nicht in die erwähnte Komponente fällt.
Damit wurde gezeigt, dass der ungerichtete Graph für nahezu alle Knoten von jedem anderen Knoten aus erreichbar ist.

Algorithmus 3.2 Starke Zusammenhangskomponenten

```
Stack S
Graph G(V,E)
int markierungsNr =0;
int komponentenNr=0;
while ∀v ∈ V ∉ S do
    Wähle v ∉ S
    markierungsNr++
    Tiefensuche(v)
end while
Transponiere Graphen G zu G^T
while S ≠ ∅ do
    v = Pop(S)
    if v nicht mit komponentenNr markiert then
        komponentenNr++
        Tiefensuche_T(v)
    end if
end while

function Tiefensuche(Knoten v)
    Markiere v mit markierungsNr
    Führe Tiefensuchenalgorithmus auf G aus
    Füge v zu S hinzu
end function

function Tiefensuche_T(Knoten v)
    Markiere v mit komponentenNr
    Führe Tiefensuchenalgorithmus auf G^T aus
end function
```

Um die entsprechende Informationen zur starken Zusammenhangskomponente zu erhalten, gibt es hier unter anderen Algorithmen die Möglichkeit den Algorithmus von Kosaraju oder den Algorithmus von Tarjan zu nutzen. In dieser Arbeit wird auf den Algorithmus von Kosaraju [Sha81] zurückgegriffen, der auf einem Stack, der Tiefensuche und der Markierung besuchter Knoten aufbaut. Der Pseudocode 3.2 beschreibt das Vorgehen hierbei genauer. An

dieser Stelle sei zu beachten, dass das Hinzufügen der Knoten in den Stack in post-order Reihenfolge erfolgen muss. In diesem Kontext sei nochmal darauf hingewiesen, dass das Transponieren in diesem Algorithmus implizit in Abschnitt 2.3 durchgeführt wurde.
Am Ende des Algorithmus 3.2 gibt die *komponentenNr* neben der Anzahl der Komponenten auch die jeweilige Zugehörigkeit eines Knotens zu einer bestimmten Komponente aus.
Die Zeitkomplexität dieses Algorithmus liegt bei dieser Implementierung auf den Wikipedia Graph bei $\mathcal{O}(|V| + |E|)$. Bei dieser Gelegenheit sei nochmal darauf hingewiesen, dass es keine Lösung mit einem besseren Komplexitätsgrad gibt, da der Graph mindestens einmal traversiert werden muss. Jedoch gibt es Algorithmen (bspw. der eingangs erwähnte Algorithmus von Tarjan), die es mit nur einem Durchlauf durch die Graphstruktur lösen.

Das Ergebnis dieser Untersuchung auf starke Zusammenhangskomponenten kann in Tabelle 3.9, Abbildung 3.1 und Abbildung 3.2 betrachtet werden.

Komponentengröße	1	2	3	4	5	6-122	5.990.933
Vorkommen EN	3.585.843	2.768	953	835	93	249	1
Komponentengröße	1	2	3	4	5	6-46	94.331
Vorkommen SIM	30.634	249	63	25	14	29	1

Tabelle 3.9.: Auswertung starker Zusammenhänge

Hier wird eine drastische Abweichung zur schwachen Zusammenhangskomponente deutlich. So hat der größte zusammenhängende Teilgraph eine Größe von ca. 6 Millionen bzw. 94.331 Knoten und kommt entsprechend nur einmal vor. Dies entspricht nach Abbildung 3.1 63% aller Knoten für den EN Graph bzw. nach Abbildung 3.2 75% aller Knoten für den SIM Graph. Das heißt demnach, dass für 63% bzw. 75% aller Artikel ein gerichteter Weg zueinander existiert. Die Knoten, die keine eigene Komponente definieren und nicht Teil der erwähnten Komponente sind, machen nur einen verschwindend geringen Anteil von 14.749 Knoten, welche sich aus 249 Komponenten der Größen 2 bis 122 zusammensetzen, aus. Für den SIM Graph ergibt sich ein vergleichbares Bild, wo diese Menge nur 1.236 Knoten umfasst. Zusammenhangskomponenten, die nur aus einem Knoten bestehen und somit den Rest der Knotenmenge definieren, kommen über 3,5 Millionen bzw. 30.634 mal vor. Eine Großteil dieser einzelnen Knoten ist abermals mit der Wikipedia Umleitungsstruktur zu erklären. So verweisen Weiterleitungsartikel zwar selber auf den eigentlichen Hauptartikel, werden jedoch selbst offensichtlich nie aus anderen Artikeln heraus verwiesen. Somit können diese Artikel im Gegensatz zur schwachen Zusammenhangskomponente kein Teil einer großen starken Zusammenhangskomponente sein.

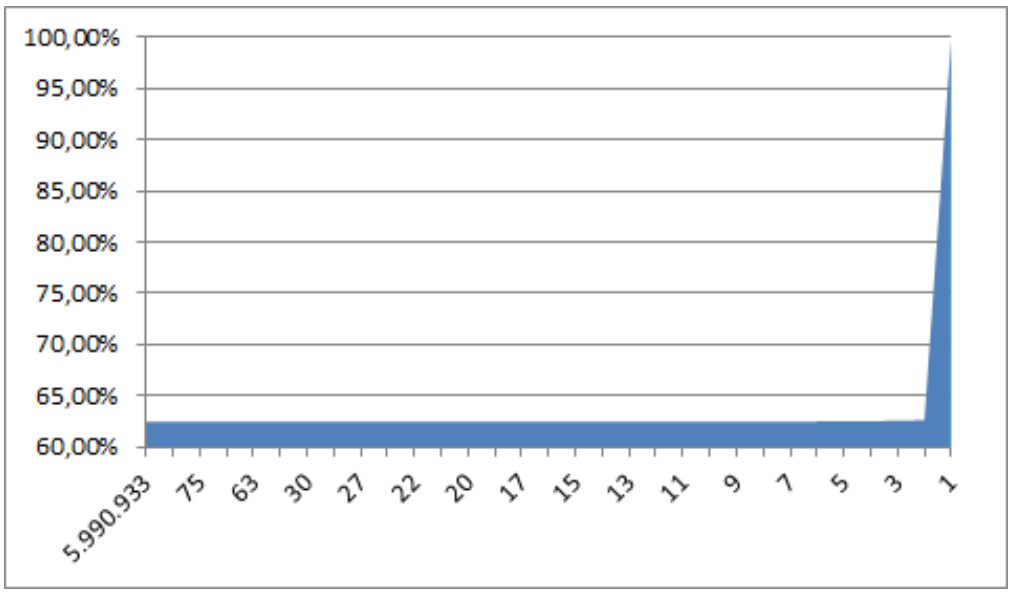

Abbildung 3.1.: Starke Zusammenhangskomponenten (EN)
x-Achse: Komponentengröße
y-Achse: Anteil an Gesamtknotenmenge

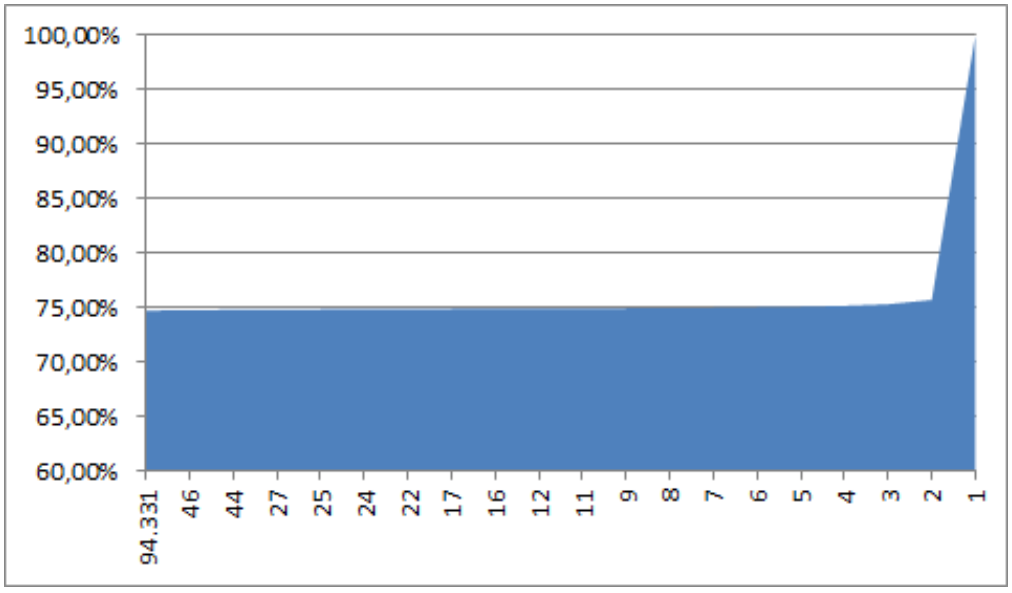

Abbildung 3.2.: Starke Zusammenhangskomponenten (SIM)
x-Achse: Komponentengröße
y-Achse: Anteil an Gesamtknotenmenge

3.3. Länge der kürzesten Wege

Um einen besseren Überblick über die kürzesten Wege zu erhalten, wird im Folgenden die Struktur der kürzesten Wege untersucht.
Dies wird mit einer modifizierten Breitensuche erreicht. Die Modifizierung besteht darin, dass die Breitensuche den Weg nicht speichert und dass jeder rekursive Aufruf der Breitensuche nur die Anzahl der Knoten für die jeweilige Tiefe bzw. Level ausgibt (das heißt das nur die Anzahl der noch nicht besuchten Nachbarn betrachtet werden).
Dass dies nicht für alle Knoten in akzeptabler Zeit möglich ist, wird unter anderem in Kapitel 4 aufgezeigt. Damit jedoch trotzdem eine Aussage darüber gefällt werden kann, wird nur eine Stichprobe der Knoten genutzt. In diesem Fall nutzt der verwendete Algorithmus 10.000 zufällig ausgewählte Knoten, was ca. 1 % der Gesamtknoten des EN Graphen darstellt.
Das Ergebnis dieser Untersuchung kann in Abbildung 3.3 bzw. Abbildung 3.4 betrachtet werden.

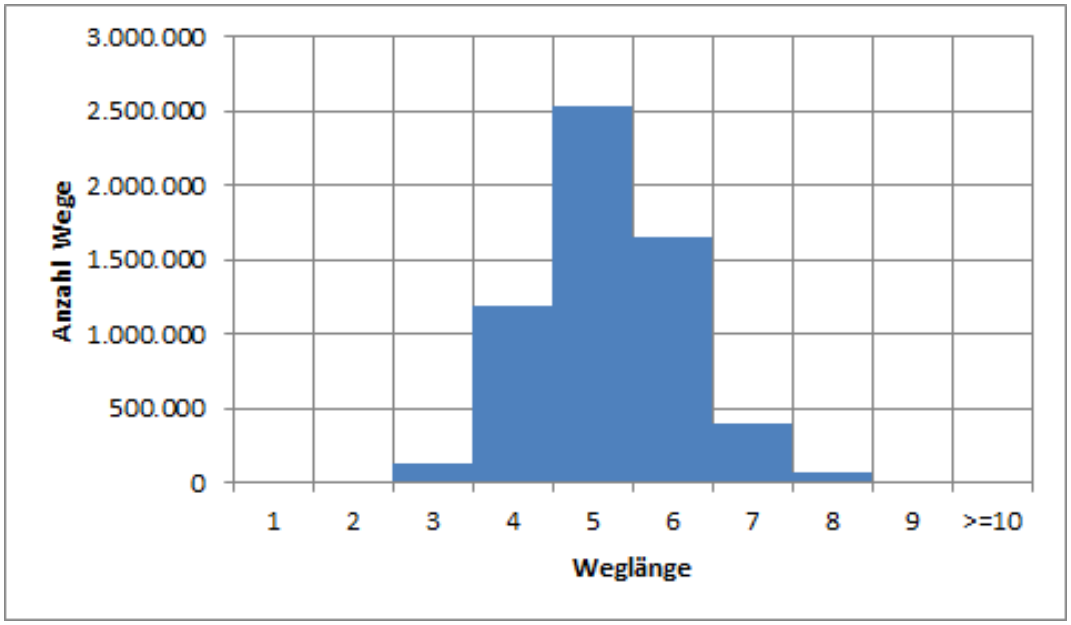

Abbildung 3.3.: Anzahl der kürzesten Wege (EN) (Stichprobengröße: 10.000)

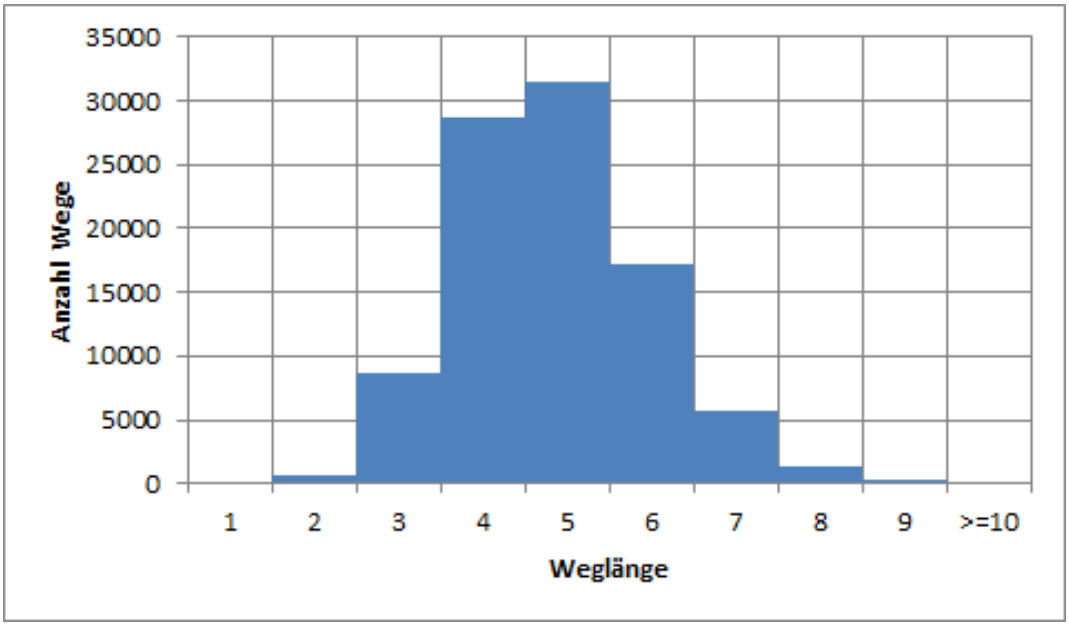

Abbildung 3.4.: Anzahl der kürzesten Wege (SIM) (Stichprobengröße: 10.000)

So ist zu erkennen, dass die meisten kürzesten Wege die Längen 3 bis 8 haben. Speziell die Wege der Längen 4, 5 und 6 machen den Großteil der kürzesten Wege (exklusive Alternativwege) aus. Im Unterschied zum EN Graph hat der SIM Graph einen größeren Anteil an kürzeren Wegen. Diese Abbildungen bestätigt wiederum die starke initiale Aufspaltung des Graphen, die ihren Höhepunkt bei Wegen der Länge 5 erreicht. Dies bedeutet für den EN Graph, dass ausgehend von einem beliebigen Knoten, es im Mittel über 2.5 Millionen Knoten der Distanz 5 gibt. Demnach besteht im Mittel ein kürzester Weg ausgehend von einem beliebigen Artikel zu einem anderen beliebigen Artikel aus nur 5,2 Abschnitten. Im SIM Graph liegt durch die in Abbildung 3.4 gezeigte Aufspaltung ein Durchschnitt der Weglänge bei 4,8 Abschnitten.
Diese Aussage ist natürlich nur als Approximation zu verstehen, da beispielsweise nicht jeder Artikel wirklich mit jedem weiteren Artikel über einen Weg verbunden ist. Es werden hier schnell Grenzen erreicht, bei denen der Versuch des Herausschreibens aller Wege bestimmter Länge nicht durchführbar wird (beispielsweise würden kürzeste Wege der Länge 5 über 25 Billionen Wege exklusive Alternativwege liefern).
Darüber hinaus wurde bei dieser Untersuchung keine weiteren kürzesten Wege als die der Länge 1.365 für den EN Graph und 17 für den SIM Graph gefunden, welche im Vergleich zu einem Straßengraph sehr gering ist. Durch eine genauere Betrachtung der Wege größer 1.000 fällt auf, dass der Zielknoten immer im Bereich bestimmter Artikel, welche entweder Asteroiden oder Asteroidenlisten sind, auftaucht. Eine genauere Untersuchung hat gezeigt, dass es 2 Arten von Asteroidenlisten gibt. Die erste Listenart zählt die Asteroiden in 100er schritten auf, und die zweite Listenart zählt die Asteroiden in 1.000er Schritten auf. So sind

die Daten redundant in 2 Artikeln aufgelistet, wobei die 1.000er Artikelliste von außen über übergeordnete Listen verlinkt werden, wodurch ein Ziel leicht gefunden wird. Bei den 100er Listen besteht keine Verlinkung über einen übergeordneten Index, sondern nur eine Verlinkung zur vorherigen oder nachfolgenden Asteroidenliste (existierende Verknüpfungen auf andere Artikel, die nicht teil der Asteroidenauflistung sind, seien hierbei nicht beachtet). Dies bedeutet, dass ein Weg zu einer Asteroidenliste erst alle vorherigen 100er Listen durchlaufen muss, sodass dadurch lange Wege entstehen. Ein Beispiel für solch einen Weg ist im Anhang unter Abbildung A.1 einzusehen.

3.4. Kantenminderung

Die Beobachtung in Abschnitt 3.3 liegt nahe, den Graph auf einfach verkette Wege zu untersuchen. Diese Untersuchung gibt Aufschluss darüber, ob solche Ketten den Graph unnötig in die Länge ziehen und somit eine Kantenkontraktion angemessen ist. Eine Wegkette sei definiert als ein Weg, bei der die Route sich aus Knoten bei der der Eingangs- und Ausgangsgrad je kleiner gleich 1 ist, zusammensetzt. Hierbei werden die folgenden Knotenarten untersucht:

- Knoten, die keine eingehenden Kante und nur eine ausgehende Kante haben.
- Knoten, die einen Ausgangsgrad größer gleich 2 haben.

Der Algorithmus 3.3 beschreibt die verwendete Methode genauer. Zu beachten gilt, dass im Unterschied zu Knoten ohne eingehende Kanten (im Algorithmus *zähleKettenEinfach*) bei Knoten mit Knotengrad größer gleich 2 die ausgehenden Knoten auf Ketten untersucht werden (im Algorithmus *zähleKetten*). Falls das Auftreten und vor allem die Länge solcher Ketten es rechtfertigt, bietet sich eine Kontaktion solcher Ketten an um so die Komplexität des Graphen zu verringern.

Das Ergebnis dieses Algorithmus wird in Tabelle 3.10 dokumentiert.

	Anzahl für ausg. Kanten		Anzahl für eing. Kanten	
Länge	EN	SIM	EN	SIM
1	3.303.516	16.838	593	108
2	25.322.701	212.015	6.029.452	139.124
3	9.410	1.501	406.833	17.290
4	400	102	36.649	784
5	4	0	4.283	71
6	0	0	358	8
7	0	0	22	0

Tabelle 3.10.: Ergebnis der Untersuchung auf Kontraktionsmöglichkeiten

Algorithmus 3.3 Untersuchung auf Kontraktionsmöglichkeiten

```
initialisiere distanzZähler[]
for all v ∈ V do
    eKnoten[] = v.eingehendeKnoten()
    aKnoten[] = v.ausgehendeKnoten()
    if eKnoten.length == 0 & aKnoten.length==1 then
        distanzZähler[zähleKettenEinfach(v)]++;
        fahre mit nächstem v fort
    end if
    if aKnoten.length<2 then
        mache mit nächstem v weiter
    end if
    for all w ∈ aKnoten do
        distanzZähler[zähleKetten(v,w)]++;
    end for
end for

function zähleKettenEinfach(v)
    int distanz = 0
    initialisiere boolean besucht[]
    besucht[startKnoten] = true
    aKnoten[] = v.ausgehendeKnoten()
    while aKnoten == 1 & !besucht(aKnoten) do
        distanz++
        besucht[aKnoten] = true
        aKnoten = aKnoten.ausgehendeKnoten()
    end while
    return distanz
end function

function zähleKetten(startKnoten, betrachteterKnoten)
    int distanz = 0
    initialisiere boolean besucht[]
    besucht[startKnoten] = true
    besucht[betrachteterKnoten] = true
    aKnoten[] = w.ausgehendeKnoten()
    while aKnoten == 1 & !besucht(aKnoten) do
        distanz = (distanz == 0) ? 2 : distanz + 1;
        besucht[aKnoten] = true
        aKnoten = aKnoten.ausgehendeKnoten()
    end while
    return distanz
end function
```

So fällt für beide Graphen auf, dass es keine langen Wegketten gibt. Das Maximum für den EN Graph liegt bei der Länge 7 und mit einem Vorkommen von 22 tritt es selten auf. Der Großteil der Wegketten sind solche der Längen 1 und 2. Die erhoffte Möglichkeit auf Kontraktion lange Wegketten und der daraus folgenden Vereinfachung der Graphstruktur stellt somit keine Option dar.

4. Breitensuche

Edward Forrest Moore hatte in den 1950er Jahren das Problem, die Leitungswege von Telefonverkehr möglichst effizient zu lösen. Um diese Anforderung bearbeiten zu können, hat Moore zuallererst einen ungerichteten Graphen mit keiner Längenfunktion als Basis genommen. In diesem Graph sollte ein kürzester Weg von Knoten A zu Knoten B so gefunden werden, sodass nur eine minimale Anzahl an Kanten genutzt werden.
In seinem Artikel zur Suche kürzester Wege in einem Labyrinth [Moo59] hat er für dieses Problem 4 Algorithmen vorgeschlagen, womit er heute als Erfinder der Breitensuche gilt. Diese beschriebenen Algorithmen lösen das Problem auf ähnliche Weise, aufbauend auf folgendem grundlegendem Vorgehen:

1. Gebe dem Startknoten das Label k=0.
2. Für alle k=0,1,...: Gebe allen benachbarten Knoten zu k, die mit keinem Label markiert sind, das Label k+1.
3. Ende falls Zielknoten erreicht wurde oder der Graph keine weiteren unmarkierten Knoten enthält

Heutzutage hat die Breitensuche neben der ursprünglichen Anwendung nach der Suche kürzester Wege noch weitere bedeutende Anwendungsfelder hinzugewonnen. Beispielsweise sind im Folgenden nur einige Anwendungsmöglichkeiten der Breitensuche aufgezeigt:

- Finde Zusammenhangskomponenten (siehe auch Abschnitt 3.2).
- Erkennen von bipartiten Graphen.
- Diverse Anwendungen in der Lexikografie.
- Copying Collection, auch Cheney's Algorithm genannt, welches eine Möglichkeit zur Garbage Collection darstellt.
- Maximalen Fluss nach dem Ford-Fulkerson Algorithmus berechnen.
- Transformation einer symmetrisch, dünnbesetzten Matrix in eine Bandmatrix mit einer geringeren Bandbreite, auch als Cuthill-McKee-Algorithmus bekannt.

4.1. Implementierung

Um die eingangs beschriebene Abarbeitung der entdeckten Knoten nach dem First In – First Out Prinzip zu ermöglichen, wird hierbei auf eine Queue zurückgegriffen. Des Weiteren wird eine besucht Markierung der Knoten benötigt, die mit einem simplen boolean Array realisiert wird. Zu guter Letzt wird für die Wegausgabe eine weitere Datenstruktur benötigt, die die Vorgängerknoten speichert. Dies wird ebenfalls mit einem Array aus dem primitiven Datentyp int gelöst, in der die Vorgänger Knoten Id gespeichert wird.
Der Algorithmus 4.1 zeigt die Vorgehensweise.

Algorithmus 4.1 Breitensuche

```
function BFS(Graph G,Startknoten,Zielknoten)
    Queue q
    boolean[] besucht = {false, ...}
    int[] vorg ={Integer.MAX_VALUE ...}
    besucht[Startknoten] = true
    vorg[Startknoten] = -1
    q.add(Startknoten)
    while !q.isempty do
        aktKnoten = q.remove()
        if aktKnoten == Zielknoten then return
        end if
        for all Nachbarkanten k zu aktKnoten do
            if !besucht[k] then
                besucht[k] = true
                vorg[k] = aktKnoten
                q.add(k)
            end if
        end for
    end while
end function
```

Mit dieser Implementierung und der Graphrepräsentation beschrieben in Abschnitt 2.2 hat dieser Algorithmus eine Laufzeit von $\mathcal{O}(|V| + |E|)$, da jeder Knoten und jede Kante im schlechtesten Fall durchsucht werden. Im schlimmsten Fall wird somit eine Laufzeit von $\mathcal{O}(|V|^2)$ erreicht, falls der Graph vollständig ist.
In Java ist diese Implementierung zwar für einen einzelnen Aufruf geeignet, wird jedoch bei erneuten Aufrufen schnell ineffizient, da im Gegensatz zu anderen Programmiersprachen keine explizite Kontrolle über die Speicherverwaltung gegeben ist. Das heißt, dass bei vielen Aufrufen jedes Mal die nötigen Datenstrukturen entweder komplett neu allokiert werden (wie in Algorithmus 4.1) oder eine Neuallokation zu einer bestehenden Datenstruktur durchgeführt wird. Beispielsweise dauert eine Allokation zu einer bestehenden Datenstruktur für ein einzelnes Array des primitiven Datentyps int der Größe $|V|$ angewandt auf alle

Knoten (entspricht etwa 9,6 Millionen Allokationen) etwa 12 Stunden und bringt teilweise eine starke CPU-Auslastung mit sich, da der Garbage Collector oft ausgeführt wird. Damit die Allokation aufeinander folgender Aufrufe effizient gelöst wird, wird folgende Alternative angewandt:

1. Die nötige Datenstruktur wie das *besucht* Array wird nur einmalig am Anfang, das heißt außerhalb vom Breitensuche Algorithmus, allokiert.
2. Zwei weitere Hilfsstrukturen werden benötigt: Einmal ein *verändert* Array mit dem primitiven Datentyp int der Größe $|V|$ und ein Integer *anzahlVeränderungen*.
3. Bei jeder Änderung von *besucht* wird ein Eintrag in *verändert* an Position *anzahlVeränderungen* mit dem Wert (entspricht der Knoten-ID) des veränderten Knotens gesetzt und *anzahlVeränderungen* wird um 1 erhöht.
4. Bei einem nötigen Reset (Neuaufruf der Breitensuche) des *besucht* Arrays wird nun anstatt einer Neuallokation die Werte, die geändert worden sind, zurückgesetzt. Dies geschieht mit Hilfe des *verändert* Arrays, in dem alle Änderungen der *besucht* Werte eingetragen sind. Im selben Schritt wird auch der Wert des *verändert* Integers auf 0 zurückgesetzt. Nach dem beide Arrays ihren initialen Zustand erreicht haben, wird *anzahlVeränderungen* zurückgesetzt.

Mit diesem Vorgehen wird die kostspielige Allokation und des daraus folgenden häufigen Ausführens des Garbage Collectors umgangen.

Um eine Routenführung zu ermöglichen, wird in Algorithmus 4.1 auch ein Vorgänger Array angelegt, welches für jeden betrachteten Knoten den Vorgänger, das heißt den Knoten, der um 1 näher am Startknoten liegt, geliefert. Falls nur eine Distanz benötigt wird, kann entsprechend ein Distanz Array angelegt werden, der für jeden betrachteten Knoten die Distanz zum Startknoten ausgibt.

4.2. Benchmark

Um eine Basis für einen späteren Vergleich zu schaffen, wird dieser Abschnitt die unterschiedlichen Zeiten, die für die Breitensuche nötig sind, liefern. Hierbei muss nach der Weglänge differenziert werden, denn kurze Wege, für die im Vergleich zu langen Wegen viel weniger Kanten untersucht werden müssen, benötigen dementsprechend kürzere Zeiten. Was jedoch auf den ersten Blick nicht auffällt, ist die Tatsache, dass Wege gleicher Länge, vor allem für längere Weglängen, auch relativ starke unterschiedliche Zeiten brauchen können. Die Erklärung für dieses Phänomen liefert Abbildung 3.3. Es ist trivialerweise ein großer Unterschied, ob der gesuchte Knoten am Anfang oder am Ende der betrachteten Nachbarknotenmenge liegt (und damit die Anzahl der Untersuchungen im Breitensuche Algorithmus stark beeinflusst).

Um eine Vergleichsbasis zu schaffen, wird diese Arbeit eine zufällige Knotenmenge auswählen, die aus 10.000 Knotenpaaren besteht. Auf diese Knotenpaare, für die nicht zwingend

eine Route existieren muss, wird der Breitensuche Algorithmus angewandt. Das Ergebnis kann in Tabelle 4.1 betrachtet werden.

Weglänge	2	3	4	5	6	7	8	≥9	∞
∅ Lfz. [ms]	1,25	34	323	952	2110	3750	4720	4820	4950
Anteil	0,0%	1,4%	12,5%	27,0%	17,2%	3,9%	0,7%	0,3%	37,0%
(ohne Wege der Länge ∞)	0,1%	2,2%	19,8%	42,9%	27,3%	6,2%	1,1%	0,4%	-

Tabelle 4.1.: Anfragezeiten zur Breitensuche (EN)

So wird auf den ersten Blick das Triviale deutlich, denn desto weiter ein Knoten vom Startknoten entfernt ist, desto mehr Zeit wird für die Suche nach dem kürzesten Weg benötigt. Kürzeste Wege Anfragen, die eine Distanz kleiner 3 haben, werden in kurzer Zeit berechnet. Ab Distanzen der Länge 4 ist die Breitensuche stark aufgespannt, so dass eine Anfrage für Wege der Länge 4 im Durchschnitt um eine Größenordnung länger dauert, als für die der Länge 3. Dieser Anstieg der Laufzeit setzt sich bis zu Wegen der Länge 8 fort. Da die Aufspannung des Graphen bzw. die Anzahl der unbesuchten Nachbarknoten bei Distanzen größer 8 gering ist (siehe in diesem Zusammenhang auch Abbildung 3.3), ändert sich die Laufzeit für Distanzen größer 8 nur geringfügig.

Weglänge	2	3	4	5	6	7	8	≥9	∞
∅ Lfz [ms]	0,1	1,03	4,62	11,85	22,22	31,85	41,2	44	44
Anteil	0,6%	7,0%	22,8%	25,2%	13,1%	4,3%	1,0%	0,3%	25,7%
(ohne Wege der Länge ∞)	0,8%	9,4%	30,7%	33,9%	17,6%	5,9%	1,3%	0,4%	-

Tabelle 4.2.: Anfragezeiten zur Breitensuche (SIM)

Die selbe Untersuchung auf den SIM Graph zeigt nach Tabelle 4.2 vor allem, dass die Laufzeiten sehr viel kürzer sind. So hat eine Breitensuche in dieser Untersuchung immer ein Ergebnis innerhalb von 44ms geliefert. Dies muss selbstverständlich auf die viel geringere Größe des Graphen und der Kanten zurückgeführt werden.
So braucht ein Durchlauf des ganzen erreichbaren EN Graphen von einem Startknoten aus im Schnitt 5 Sekunden, welche in der Tabelle 4.1 als Wege der Länge unendlich gekennzeichnet sind. Bei dieser Untersuchung, in der die Knotenpaare zufällig gewählt worden sind, hatten nach mehrere Durchläufen im Schnitt 35-39% der Knotenpaare keine verfügbare Route. Der Großteil der kürzesten Wege hat die Längen 4 - 7, die 61% aller Anfragen bzw. 96% aller erfolgreichen Anfragen ausmachen. Für den SIM Graph ergibt sich ein leicht verändertes

Bild. So ist auch hier der Großteil der kürzesten Wege in einem Bereich, jedoch startet dieser Bereich schon bei Wegen der Länge 3 und endet bei 7, womit 72% aller Anfragen bzw. 97% aller erfolgreichen Anfragen abgedeckt werden. Der Anteil der Wege ohne Route deckt sich sowohl für EN als auch für SIM mit der Untersuchung in Abschnitt 3.2, bei der für die Knotenpaare, die in der großen starken Zusammenhangskomponente liegen, ein Weg gefunden wird (siehe Abbildung 3.1 und Abbildung 3.2)
Bei genauerer Betrachtung der Ergebnisse fällt jedoch auf, dass die Laufzeiten für Distanzen größer als 3 sehr stark variieren können. So ergeben sich für bestimmte Knotenpaare, bei denen die Anzahl der Nachbarknoten und damit die Anzahl der Vergleiche relativ gering sind, Laufzeiten, die nur im Bereich von einem Zehntel der durchschnittlich berechneten Zeiten liegen.
Abschließend bleibt nur noch zu zusammenfassen, dass die worst case Laufzeiten der Breitensuche angewandt auf den englischen Wikipedia-Linkgraph im Bereich um 5 Sekunden und für den Simple English Wikipedia-Linkgraph im Bereich um 45 Millisekunden liegen.

5. Beschleunigungstechniken

Um die grundlegende Frage nach einem kürzesten Weg in einem Graph mit nicht negativen Kantengewichten startend von einem Punkt A nach Punkt B lösen zu können, wird auch heute noch der im Jahr 1959 veröffentlichte Algorithmus 5.1 von Edsger W. Dijkstra [Dij59] verwendet.

Algorithmus 5.1 Dijkstra Algorithmus für Graph G=(V,E) und s-t-Weganfrage

```
for all v ∈ V do vorganger[v] ← ∅
end for
for all v ∈ V\{s} do vorganger[v] ← ∅
end for
dist[s] ← 0
while es existiert unmarkiertes v mit dist[v] < ∞ do
    Markiere v mit minimaler Distanz
    if v == t then
        return dist[t] und kürzesten Weg
    end if
    for all e = (v, w) ∈ E do
        if dist[v] + c_e < dist(w) then
            dist[w] ← dist[v] + c_e
            vorganger[w] ← v
        end if
    end for
end while
return Kein Weg gefunden
```

Für einen gegebenen Start- und Zielknoten nutzt die ursprüngliche Implementierung dieses Algorithmus als Datenstruktur Arrays, womit eine Anfrage nach dem kürzesten Weg in einer worst case Laufzeit von $\mathcal{O}(|V|^2)$ beantwortet werden kann (Minimum entfernen $\mathcal{O}(|V|)$, Einfügen $\mathcal{O}(1)$, Wert neu einfügen $\mathcal{O}(1)$, Initialisierung $\mathcal{O}(|V|)$). Der Algorithmus wurde 1984 von Fredman und Tarjan [FT84] mit Hilfe einer Prioritätswarteschlange implementiert, so dass die worst case Laufzeit auf $\mathcal{O}(|E| + |V| \log |V|)$ optimiert werden konnte (Minimum entfernen $\mathcal{O}(\log |V|)$, Einfügen $\mathcal{O}(1)$, Wert neu einfügen $\mathcal{O}(1)$, Initialisierung $\mathcal{O}(1)$).
Um ein besseres Bild von den Laufzeiten zu erhalten, braucht dieser Algorithmus in der Praxis angewandt auf den Europa Graphen mehrere Sekunden, um ein Ergebnis zu liefern. In der heutigen Zeit sind solche Zeiten der breiten Masse der Nutzer selbstverständlich nicht mehr zuzumuten. Daher wurden diverse Methoden erforscht, die eine Beschleunigung

dieses Prozesses erlauben, sodass Nutzer Ergebnisse teilweise im Mikrosekunden Bereich erhalten.

Um eine solche Beschleunigung der kürzeste Weg Anfragen zu erreichen, gibt es im Grunde 2 Möglichkeiten (die sich nicht gegenseitig ausschließen). Entweder es wird versucht das Problem zu parallelisieren, oder es werden optimierte Algorithmen verwendet, die sich in eine Vorberechnungs- und eine Anfragephase aufteilen.
Bei der ersten Möglichkeit wäre hier ein Divide&Conquer Ansatz denkbar, bei dem bestimmte Teilstrecken entsprechend auf verschiedene Rechner aufgeteilt werden. Hierbei ergeben sich jedoch neben den allgemeinen Problemen von Parallelisierungen auch die theoretische Frage, in wie weit eine Anfrage aufgesplittet werden kann und ob am Ende der Geschwindigkeitsgewinn den hohen Synchronisationsaufwand der Parallelisierung rechtfertigt. Um einen Nutzen aus der Parallelisierung ziehen zu können, sollte diese im Vergleich zur zweiten Möglichkeit der Beschleunigung einen messbaren Zeitvorteil bieten. Da diese Algorithmen teilweise Ergebnisse im Mikrosekunden Bereich liefern, besteht Zweifel daran, ob ein parallelisierter Algorithmus effektiv für eine einzelne Kürzeste Weg Anfrage geeignet ist. Die im folgenden aufgezeigten Beschleunigungsverfahren sollen nur grundlegend zeigen, welche Algorithmen es zur Auswahl gibt (ohne Anspruch auf Vollständigkeit) und wie diese arbeiten. Der Einfachheit halber sei für jeden folgenden Algorithmus ein Graph G=(V,E) gegeben, worauf eine kürzeste Weg Anfrage d(s,t) erfolgt.

5.1. Vorberechnung aller Routen

Nach der eingangs erwähnten Erläuterung stellt sich die berechtigte Frage, wieso nicht einfach alle Routen berechnet und serialisiert werden. Damit dürfte jede Anfrage, sei es nun eine kurze oder lange Route, in konstanter Zeit ein Resultat liefern, da jedes Ergebnis schon vorliegt und die Lösung im Grund aus einem Aufruf dieser vorberechneten Lösungen besteht.
Dies bedeutet, dass $\mathcal{O}(|V|^2)$ Anfragen beantwortet und die entsprechenden Routen gespeichert werden müssten. Für den Wikipedia Graph mit 9,6 Millionen Knoten würde dies etwa 10^{12} Anfragen bedeuten.
Tabelle 5.1 zeigt einen praktischen Versuch, bei der nur Routen (ohne Alternativrouten) bestimmter Länge berücksichtigt wurden. Um zu sehen, wie umfangreich die Daten werden, wird der Versuch jeweils mit und ohne Schreiben der Daten und mit einem und mehreren Threads wiederholt. Die Länge eines Weges sei definiert als die Anzahl besuchter Knoten ab dem Startknoten (beispielsweise hätte die Route $Start\ v_1\ v_2\ v_3$ die Länge 3). Wege der Länge 1, dass heißt alle Knoten, die über eine ausgehende bzw. eingehende Kante direkt mit einem weiteren Knoten verbunden sind, seien trivialerweise in dieser Untersuchung ausgelassen.
Es wird schnell ersichtlich, dass die Theorie durch die Praxis bestätigt wird (das Format der Serialisierung der Routen ist $Start\ v_1\ v_2\ ...\ v_x \backslash n$ und wird in einer unkomprimierten Textdatei gespeichert). Falls ausschließlich der SIM Graph betrachtet wird, so wird deutlich, dass sowohl die Berechnungszeiten wie auch der Platzverbrauch im Rahmen des Möglichen liegen. So sind Wege der Länge 2 bis 4 in zusammen unter einer Stunde vorberechnet und der benötigte Platz hält sich mit ca. 132GB ebenfalls in Grenzen. Mit Hilfe der Abbildung 3.4 lässt sich erschließen, dass Wege der Länge 5 nur etwas zahlreicher als Wege der Länge 4 sein dürften (laut Abbildung wären dies grob $31.000 * 126.201 = 3.9 \times 10^{12}$).

	EN			SIM		
Länge der kürzesten Wege	2	3*	4**	2	3	4
Anzahl kürzester Wege	24×10^9	$1,2 \times 10^{12}$	11×10^{12}	79×10^6	$1,1 \times 10^9$	$3,6 \times 10^9$
∅ Anzahl kürzester Wege pro Knoten	2.506	125.110	1.146.845	624	8.662	28.685
Zeit [h] inkl. Speichern (1 T)	3,47	192,88	3.359,79	0,01	0,15	0,65
Zeit [h] inkl. Speichern (6 T)	1,76	110,53	1.654,83	0,00	0,10	0,31
Zeit [h] ohne Speichern (1 T)	0,92	69,80	1321,31	0,00	0,03	0,22
Zeit [h] ohne Speichern (6 T)	0,28	21,57	510,11	0,00	0,01	0,05
Platzverbrauch [GB]	551	37.000	487.000	1,4	26	105

Tabelle 5.1.: Extraktion kürzester Wege
* Hochrechnung auf Basis von 100.000 Knoten
** Hochrechnung auf Basis von 10.000 Knoten

Somit ist die vollständige Vorberechnung der Wege zwar möglich, es stellt sich jedoch die Frage, ob dies mit den gemessenen Zeiten für die Breitensuche in Abschnitt 4.2 gerechtfertigt ist. Hier sei nochmals darauf hingewiesen, dass die Berechnung nur einen kürzesten Weg betrachtet und etwaige Alternativwege ausgelassen wurden.
Für den um 2 Größenordnungen mächtigeren EN Graph ergibt sich ein stark unterschiedliches Bild. So sind kürzeste Weg Anfragen der Länge 2 mit 551GB Speicherplatz und 1,5 bis 3,5 Stunden Berechnungszeit noch realisierbar. Für Anfragen der Länge 3 ist die Rechenzeit zwar noch vertretbar, der Speicherbedarf wächst jedoch exponentiell. Spätestens bei kürzesten Wegen der Länge 4 kommen Werte auf, die so nicht mehr effizient bearbeitet werden können. Hierbei fällt auf, dass das Multi-Threading, in der Daten auf die Festplatte geschrieben werden, nur einen vergleichsweise geringen Geschwindigkeitsvorteil mit sich bringt. Dies ist mit der erhöhten Anzahl der Schreibzugriffe auf der Festplatte zu begründen, da mehrere Threads die Festplatte voll auslasten, sodass einige Threads lange Zeit still stehen um auf die Festplatte schreiben zu können. Wo jedoch viel Zeit in Berechnungen investiert wird, beispielsweise bei Wegen der Länge 4, ist diese Parallelisierung entsprechend größer von Vorteil.

Mit dieser Untersuchung wurde anschaulich gezeigt, wieso eine Vorberechnung aller kürzester Wege für alle Knoten (ohne Alternativwege) zu zeit- und speicherplatzintensiv ist. Zusätzlich müsste bei jeder Änderung dieser Vorgang wiederholt werden.

5.2. Bidirektionale Suche

Die Bidirektionale Suche arbeitet wie der Dijkstra Algorithmus eine kürzeste Weg Anfrage ab. Im Gegensatz dazu startet dieser Algorithmus 2 Anfragen. Die erste Anfrage startet von Startpunkt s und die zweite Anfrage startet vom Zielpunkt t, jedoch auf Basis der eingehenden Kanten. Durch die 2 Suchen hat dementsprechend jeder Knoten zwei Distanzwerte und zwei Vorgängerarrays. Sobald sich diese Anfragen in einem gemeinsamen Punkt u treffen, wird die Route zusammengeführt und das Ergebnis als die Distanz von s zu u und der Distanz von t zu u geliefert. Für eine Weganfrage müssen die entsprechenden Vorgängerarrays des jeweiligen Suchraums genutzt werden.

Durch die Aufteilung der Suche in 2 kleinere Teilräume wird die betrachtete Fläche, welche der Anzahl der Knoten entspricht, auf

$$\frac{2 * \pi(\frac{r}{2})^2}{\pi r^2} = \frac{1}{2}$$

verringert. Der verkleinerte Suchraum, die eine Verringerung der Komplexität der Anfrage bedeutet, erlaubt eine schnellere Suche vor allem durch eine parallele Implementierung der Anfrage. Existiert jedoch kein s-t-Weg, so wird der Aufwand im Vergleich zum klassischen Dijkstra verdoppelt.

Angewandt auf die Breitensuche im Wikipedia Graph würde diese Implementierung zwar den Suchraum verkleinern, jedoch würde die Tatsache, dass der Graph nicht sehr in die Tiefe geht und eine zweite Breitensuche große Initialisierungs- und Platzkosten verursacht, nur bedingt einen Vorteil bringen.

5.3. A*-Algorithmus

Der A*-Algorithmus, der häufig auch als Erweiterung und Verallgemeinerung des Dijkstra Algorithmus aufgefasst wird, wurde erstmalig im Jahr 1968 von Hart et al. [HNR68] beschrieben und findet immer die optimale Lösung. Die grundlegende Erweiterung von diesem Algorithmus ist die Nutzung einer Zielheuristik, womit eine Verringerung der Laufzeit erreicht wird. Das heißt, dass zuerst nur bestimmte Knoten, die zu diesem Zeitpunkt mit größter Wahrscheinlichkeit auf der optimalen Route liegen, in der Suche berücksichtigt werden. Das bedeutet, dass für einen Knoten u auf dem s-t-Weg der nächste Knoten mit dem Minimum aus der Gleichung

$$f_{s,t}(u) = c_{s,t}(u) + h_{s,t}(u)$$

gewählt wird, wobei $c_{s,t}(u)$ die bisherigen und $h_{s,t}(u)$ die verbleibenden, geschätzten Kosten vom Knoten u zum Zielknoten t darstellen. Die berechneten Kosten $f_{s,t}(u)$ werden in einer Prioritätswarteschlange eingefügt und entsprechend nach Bearbeitung der betrachteten Knoten der erste (und somit der auf dem potentiell kürzesten Weg liegende) Knoten gewählt. Genau wie beim Dijkstra Algorithmus kann der gefundene Weg über die Vorgängerliste heraus abgeleitet werden.
Mit dieser Methode liegt in der verwendeten Heuristik die bedeutende Rolle. Die genutzte Heuristik darf die tatsächlichen Kosten nicht überschreiten und sollte eine möglichst geeignete Schätzung verwenden. Auf einem Straßengraphen wird häufig der euklidische Abstand als Schätzfunktion verwendet, welche dank der Dreiecksungleichung stets eine optimistische Schätzung der Distanz liefert. Weitere Anwendungsgebiete sind beispielsweise das 8- und 15-Puzzle Spiel, bei welcher die Anzahl der falsch platzierten Steine als Heuristik verwendet werden kann.
Die Laufzeit entspricht der Laufzeit des Dijkstra Algorithmus. Empirische Untersuchungen von S. Hasselberg haben jedoch gezeigt, dass dank der Heuristik 2-5 mal weniger Knoten untersucht werden müssen [Hasoo].

Die Luftlinie kann bei der Suche im Wikipedia Graph offensichtlich nicht verwendet werden, womit sich die Frage in wie fern sich dieser Algorithmus auf den Wikipedia Graph anwenden lässt, auf die Lösung nach der Suche einer geeigneten Heuristik beschränkt.

5.4. Arcflags-Algorithmus

Das Konzept des Arcflags Algorithmus, erstmalig 1997 von Lauther [Lau97] und [Lauo4] publiziert, welches die darauffolgenden Jahre von diversen anderen Forschern verbessert wurde, u.a. [MSS+05], [KMS05] und [MSS+06], ist ein weiteres zielgerichtetes Beschleunigungsschema. Hierbei wird versucht, die Menge der Kanten mit Hilfe einer Vorberechnung zu minimieren, um damit eine Beschleunigung der Anfrage zu erreichen.
In der Vorberechnungsphase wird der Graph in Zellen mit möglichst ähnlich großer Knotenkardinalität partitioniert und jeder Knoten wird einer Zelle C zugewiesen. Ein Bit Vektor f_e mit der Größe $|C|$, welches als *flag* bezeichnet wird, wird für jede Kante angelegt, wobei jedes Bit mit der Zelle C_i korrespondiert. Dieser Bit Vektor $f_e(i)$ für Kante e wird jeweils nur dann auf 1 gesetzt, wenn e auf mindestens einer kürzesten Route liegt, bei welcher der Zielknoten in C_i liegt. Anfänglich sind alle Bits bis auf Einen auf 0 gesetzt. Das einzige Bit, welches auf 1 gesetzt wird, ist die korrespondierende Zelle, zu welcher die Kante gehört. Im nächsten Schritt werden die flags entsprechend belegt. Dabei wird überprüft, ob startend von einem beliebigen Knoten zu einem Knoten in Zelle C_i die Kante auf der kürzesten Route liegt. Ist dies der Fall, wird das entsprechend Bit auf 1 gesetzt. Um für diese Anfrage nicht die komplette Knotenmenge betrachten zu müssen, wird diese Berechnung folgendermaßen optimiert.

Es werden zuerst alle Knoten bestimmt, die eine Kante zu einer Nachbarzelle haben. Von diesen Grenzknoten aus wird der kürzeste Weg Baum zu allen Knoten außerhalb der eigenen Zelle aufgespannt. Falls nun eine Kante e Teil eines kürzesten Wegs zu einem Grenzknoten in der Zelle C_i ist, wird das flag der Kante e $f_e(i)$ auf 1 gesetzt.
Ein wichtiger Faktor für diese Vorberechnung ist wie der Graph partitioniert wird. Der intuitive Weg den Graphen mit Hilfe einer geographischen Gitterstruktur aufzuteilen funktioniert gut, jedoch kommen [KMS05] und [MSS+05] auf das Ergebnis, dass der *multi-way arc seperator*, welches hier nicht weiter aufgeführt werden soll, hervorragende Resultate liefert. Eine weitere auf Arcflags aufbauende Verbesserung stellt der PHAST Algorithmus von Delling et al. [DGNW11] oder der SHARC Algorithmus von Bauer und Delling [BD09] dar.
Nach dem die Vorberechnungsphase abgeschlossen ist, wird eine Anfrage mit Hilfe eines leicht modifizierten Dijkstra Algorithmus gelöst. Hierbei wird der Algorithmus dahingehend verändert, so dass nur Kanten mit einem gesetzten flag Bit der Zielzelle in Betracht gezogen werden.
So wird ein Beschleunigungsfaktor im Bereich von 10^3 bis 10^4 erreicht, bei der zwar relativ wenig zusätzliche Informationen gespeichert werden müssen (pro Kante $|C|$ Bits), dem jedoch eine rechenintensive Vorberechnungsphase gegenübersteht.

5.5. Landmark-Algorithmus

Der Landmark-Algorithmus, auch ALT Algorithmus genannt (**A***, **L**andmark and **T**riangle inequality), wurde 2005 von Goldberg et al. [GH05] vorgestellt und ist eine Variante des A*-Algorithmus, bei der als Potential für die Zielheuristik Landmarks (auf Deutsch: Orientierungspunkte) und die Dreiecksungleichung verwendet werden.
Hierbei wird in der Vorberechnungsphase zuerst eine Menge $L \subset V$ von k Orientierungspunkten berechnet. Danach werden die Distanzen aller Knoten $v \in V$ von und zu jedem der Punkte in L berechnet und gespeichert. Damit gilt bei einer s-t-Anfrage für alle $s \in V$ folgende Ungleichung

$$d(s,t) \geq d(l_1,t) - d(l_1,s)$$
$$d(s,t) \geq d(s,l_2) - d(t,l_2)$$

Somit wird die folgende untere Schranke für d(s,t)

$$\underline{d}(s,t) = max_{l \in L}\{max\{d(l,t) - d(l,s), d(s,l) - d(t,l)\}\}$$

als Potential verwendet. Da die detaillierte Aufzählung der möglichen Berechnungen der Orientierungspunkte den Rahmen dieser Arbeit sprengen würde, seien die folgenden Möglichkeiten für die Berechnung als Beispiele erwähnt:

- Zufällig: Es werden zufällige Knoten als Orientierungspunkte gewählt, was eine schnelle Vorberechnung erlaubt, bei der jedoch der Beschleunigungsfaktor im Allgemeinen gering ist.
- Farthest: Orientierungspunkte so auswählen, sodass die Distanzen untereinander maximal sind. Gut geeignet für nicht geographische Graphen wie den Wikipedia Graph.
- Planar: Ein zentraler Orientierungspunkt. Der Rest wird gleichmäßig am Rand des Graphen gewählt. Funktioniert für geographische Graphen gut, schlägt jedoch auf dem Wikipedia Graph fehl da keine geographischen Informationen gegeben sind.

- Avoid: Berechne kürzesten Wege Baum T_r von einem Knoten r. Bei jeder Iteration (während dem Bau des kürzesten Wege Baums) wird für jeden Knoten v das Gewicht als Differenz von $d(v,r)$ und $\underline{d}(v,r)$ gespeichert. Falls die Summe aller Gewichte seiner Nachfolger in T_r bei o liegt, so ist mindestens ein Nachfolger in T_r ein Orientierungspunkt. Nun wird im Baum der Knoten w mit maximaler Summe der Gewichte ausgewählt und traversiert T_r. Anschließend wird startend von v immer dem Knoten gefolgt, welcher die maximale Summe besitzt. Das so erreichte Blatt wird zu der Menge L hinzugefügt.
- Lokale Optimierung: Berechne mehr Orientierungspunkte als nötig, und wähle die besten Punkte durch eine Optimierungsfunktion aus.
- MaxCover: Verknüpfung aus Avoid und lokaler Optimierung.

Durch dieses Vorgehen wird der Suchraum verkleinert und damit die Suche beschleunigt. Für Straßengraphen werden häufig 16 Landmarken verwendet, da hiermit der beste Kompromiss zwischen zusätzlichem Speicher, Vorberechnungszeit und dem erreichten Beschleunigungsfaktor erreicht wird. Als Beispiel für den Europa Graphen liegt der Beschleunigungsfaktor für ALT mit 16 Landmarken bei 10^2 (bei bidirektionaler Suche) und jeder Knoten muss zusätzlich 128 Byte an Informationen speichern [GH05]. Delling und Wagner zeigen in [DW07] wie eine effiziente Implementierung für einen dynamischen Graph aussehen kann.

5.6. Contraction Hierarchies

Contraction Hierarchies (kurz CH) ist ein weiteres Beschleunigungsschema vorgestellt von Geisberger et al. [GSSD08], welches sich bestimmte Eigenschaften eines Graphen zu Nutze macht.
Bis vor ein paar Jahren wurde die folgende grundlegende Hierarchisierung von Straßengraphen genutzt:
Bei weit voneinander entfernten Start- und Zielknoten werden häufig nur bestimmte Kanten genutzt, welche auf einem Straßengraph abhängig von der Distanz entweder Autobahnen, Bundesstraßen oder Hauptstraßen sein können. Um eine Unterscheidung für diese wichtigen Kanten durchführen zu können, werden Hierarchieebenen auf den Graph gelegt, und je nach s-t-Distanz die jeweilige Ebene betrachtet. So werden bei relativ kurzen Distanzen, beispielsweise innerhalb einer Stadt, die Ebene mit den Hauptstraßen für diese Methode in Betracht gezogen.
Mit der Einführung von CH wird die erwähnte Methode abgewandelt. Anstatt Hierarchieebenen auf den ganzen Graphen einzuführen, werden Level (auch Ordnung) für Knoten eingeführt. Bei der anschließenden bidirektionalen Suche startend von Start- und Zielknoten, werden vom Startknoten nur Kanten zu einem Knoten mit höherem Level und vom Zielknoten ebenfalls nur Kanten zu Knoten mit höherem Level betrachtet. Schlussendlich wird die gefundene Route *entpackt*, das heißt, dass die kontrahierten Kanten, die im folgenden noch beschrieben werden, auf ihre ursprüngliche Form gebracht werden.
Die Konstruktion dieser Level, aus welcher sich der Name ableitet, entsteht für einen Knoten u in dem für genau diesen der Kontraktionsschritt ausgeführt wird. Hierbei wird der Knoten aus dem Graph gelöscht, und entsprechende Kanten eingeführt, falls Informationen über einen kürzesten Weg verloren gehen.

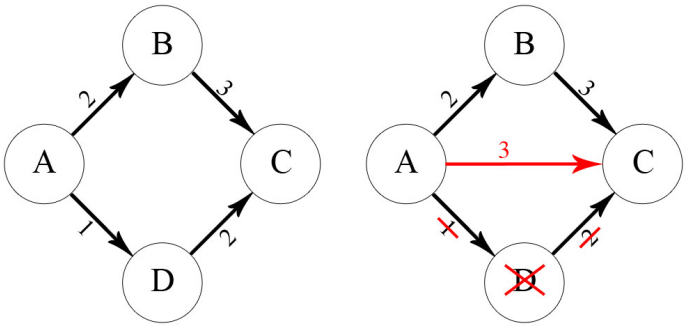

Abbildung 5.1.: Kontraktion am Beispiel

Am Beispiel von Abbildung 5.1 wird deutlich, dass bei Entfernen vom Knoten D und dessen Kanten, der kürzeste Weg von A nach C über D verloren gehen würde. Als Ausgleich wird eine Kante von A nach C mit den entsprechenden Kosten in Höhe von 3 eingefügt. Die Reihenfolge, in der die Knoten so abgearbeitet werden, wird die Knotenordnung genannt.
Bei einer s-t-Anfrage erlaubt uns die Knotenordnung und die ersetzten Kanten eine Beschränkung des Suchraums. Die Auswahl beim Kontraktionsschritt spielt hierbei eine entscheidende Rolle. Dieser Algorithmus funktioniert zwar für eine beliebige Auswahl von Knoten im Kontraktionsschritt, jedoch wird die Beschleunigung dadurch nicht so stark verbessert wie durch andere spezielle Heuristiken. Im Folgenden sei die edge difference Heuristik (kurz ED) als Beispiel erwähnt, bei der jeder Knoten in einer Prioritätswarteschlange verwaltet und entsprechend ihrer Priorität (entspricht der ED) bearbeitet wird. Die edge difference setzt sich zusammen aus der Differenz von S, welcher die Anzahl der zusätzlich einzuführenden Kanten bei einer Kontraktion darstellt, und E, welcher die Anzahl der benachbarten Kanten vom betrachteten Knoten ist. Zu beachten gilt hierbei das bei jedem Kontraktionsschritt sich die ED für die Knoten ändern weshalb diese entsprechend angepasst werden müssen. Für dieses Vorgehen gibt es ebenfalls Heuristiken, da die naive Methode der Neuberechnung für jede einzelne Kontraktion eine quadratische Laufzeit liefern würde. Beispielsweise seien hierzu die *lazy update*, *neighbours only* oder *periodic update* Heuristik genannt.

5.7. Transitknotenkonzept

Im Rahmen der ALENEX 2007 veröffentlichten Bast et al. [BFM+07] ein neues Verfahren, dass nicht wie die meisten anderen Beschleunigungstechniken auf Zielheuristiken aufbaut, sondern einen weiteren, intuitiv leicht verständlichen, Algorithmus formulierten. Dieser Ansatz versucht gezielt die Knoten in einem Graphen zu finden, über die die meisten *langen* s-t-Anfragen laufen. Intuitiv bedeutet dies, dass für eine Strecke von beispielsweise Stuttgart nach München, es zentrale Punkte gibt, über die die urbane Gegend verlassen (Autobahnauffahrten) und erneut im Zielgebiet in die urbane Gegend um München eingetreten wird.

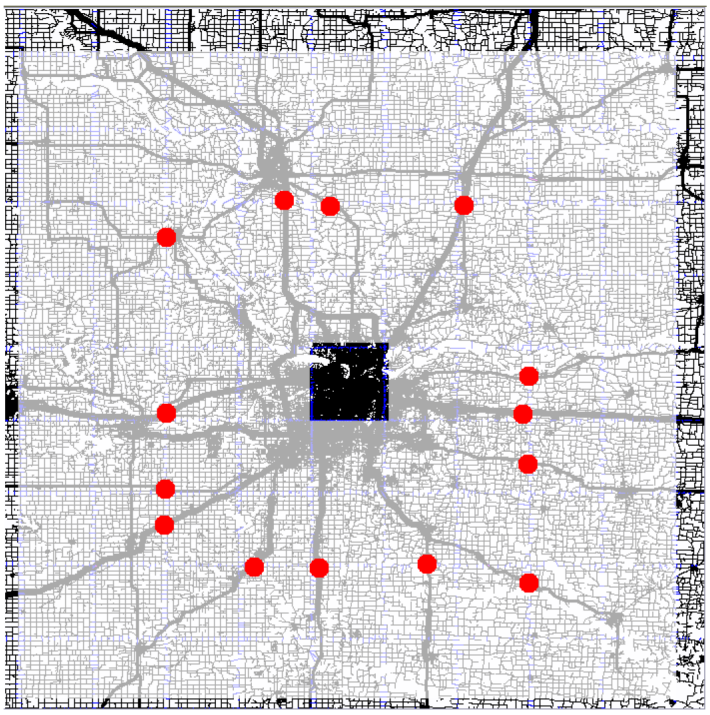

Abbildung 5.2.: Transitknoten (rot) für *lange* kürzeste Wege Anfragen (Bereich außerhalb der hellgrauen Markierung) für eine urbane Gegend (dunkler Bereich). ([BFM09], S. 2)

In Abbildung 5.2 wurde der Graph in eine Gitterstruktur aufgeteilt, und die Zugangsknoten für die dunkel markierte Zelle entsprechend rot markiert. Die Vereinigung aller Zugangsknoten definiert die Transitknotenmenge. Das heißt, dass für jede Anfrage, in der der Zielpunkt außerhalb des hellgrauen Bereichs liegt, einen der 14 rot markierten Zugangspunkte nutzt.
Das grundlegende Konzept dieses Verfahrens ist also die Aufteilung des Straßengraphen in eine Gitterstruktur, sodass für jede Zelle eine geeignete Auswahl an Zugangsknoten gefunden wird, und zeitgleich die Anzahl der Transitknoten als Menge aller Zugangsknoten so gering gehalten wird, so dass eine Berechnung und Speicherung der paarweisen Distanzen aller Transitknoten realistisch

erreichbar bleibt. Zusätzlich speichert jeder Knoten die Distanzen zu ihren Zugangsknoten, sodass eine kürzeste Weg Anfrage für hinreichend weit voneinander entfernte Punkte einfach durch

$$d(s,t) = \min\{d(s,u) + d(u,v) + d(v,t) : u \in AN(s), v \in AN(t)\}$$

zu lösen ist, welches nur ein Lesen der gesicherten Werte erfordert und damit in annähernd konstanter Zeit erfolgt.
Natürlich stützt sich der ganze Ansatz des Transitknotenkonzepts auf der Hoffnung, dass die Menge der gefundenen Transitknoten klein bleibt (im Bereich $\mathcal{O}(\sqrt{V})$), so dass eine Speicherung der paarweisen Distanz der Transitknoten einen akzeptablen Speicherverbrauch vorweist, und zeitgleich die Transitknoten eine möglichst große Menge an kürzeste Weg Anfragen abdeckt.

5.7.1. Transitknoten und Distanztabellen

Um die Menge der betrachteten Knoten gering halten zu können, wird auf dem Straßengraph eine Gitterstruktur der Größe $g \times g$ für eine Zahl g gelegt. Es sei C eine beliebige Zelle. Um die Zugangsknoten AN_C für C definieren zu können, wird ein inneres Quadrat *inner* im Abstand von 2 Zellen und ein äußeres Quadrat *outer* im Abstand von 4 Zellen um C gelegt, sodass wie in Abbildung 5.3 3 ineinander geschachtelte Quadrate entstehen.

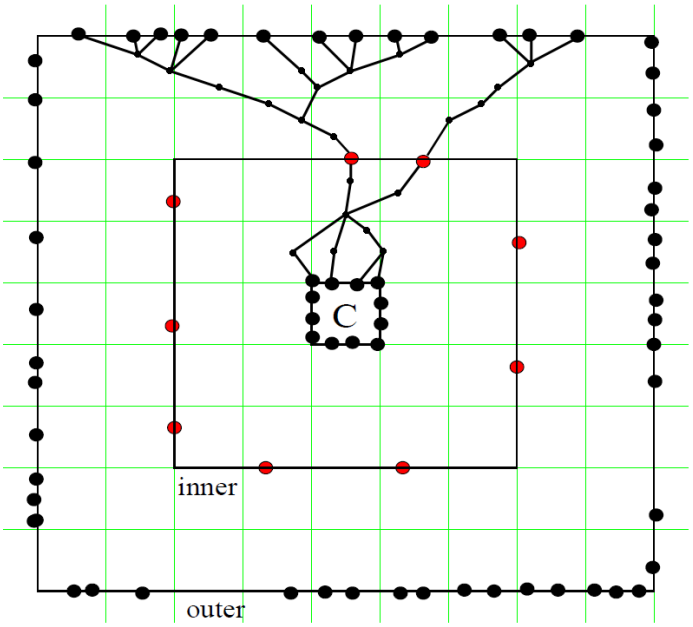

Abbildung 5.3.: Berechnung der Transitnoten.
(In Anlehnung an [BFM+07], S. 6)

Im nächsten Schritt werden die Grenzknoten V_C, V_{outer} und V_{inner} definiert. Die Konstruktion für V_C (entsprechend wird für V_{outer} und V_{inner} verfahren) erfolgt über E_C, welches die grenzüberquerenden Kanten darstellt, die also einen Knoten innerhalb von C und einen Knoten außerhalb von C haben.

Ein Endpunkt dieser beiden Knoten wird in die Menge V_C hinzugefügt. Abbildung 5.3 zeigt das Ergebnis dieses Vorgehens: V_{outer} sind die schwarzen Punkte auf outer, V_{inner} entsprechend die roten Punkte auf inner und die Punkte um C sind V_C. In der Abbildung wurde zum besseren Verständnis beispielhaft der Weg für ein paar Knoten eingezeichnet. Durch diese Konstruktion wird die Menge der zu betrachteten Knoten stark verringert, da beispielsweise für C nicht mehr alle Knoten für die folgende Berechnung in Betracht gezogen werden müssen, sondern nur die Knoten V_C, da jeder Knoten innerhalb C bei einer *langen* Weganfrage zwingend die Zelle über die definierten Knoten verlassen muss. Für die Berechnung von AN_C wird nun für jeden Knoten $u \in V_C$ eine kürzeste Weg Anfrage zu jedem Knoten $v \in V_{outer}$ gestartet. Jeder Knoten in V_{inner}, der dabei über einen dieser Wege führt, wird zu der Menge AN_C hinzugefügt. Die Transitknoten sind nun die Vereinigung aller Zugangsknoten der Zellen. Eine Optimierung dieses Verfahrens kann über einen *sweep-line Algorithmus* erfolgen, welche hier nicht weiter ausgeführt werden soll.

Während dieser Berechnungsphase speichert jeder Knoten v in einer Zelle C die Distanz zu AN_C. Nach [BFM09] benötigt der US Straßengraph bei einer 128×128 Gitterstruktur weniger als 8.000 Transitknoten, welches einer durchschnittlichen Anzahl von 11,4 Transitknoten pro Zelle entspricht. Es muss bei der Wahl der Gitterstruktur sehr genau darauf geachtet werden, ob die Gitterstruktur die optimale Lösung liefert, denn eine kleine Gitterstruktur erzeugt zwar weniger Transitknoten und in Folge dessen eine kleinere Distanztabelle, jedoch nimmt dadurch die Anzahl der lokalen Anfragen zu. Umgekehrt wird der Anteil der lokalen Anfragen bei einer dichteren Gitterstruktur sehr niedrig, jedoch wird die Anzahl der Transitknoten stark größer und damit auch die Distanztabelle.
Um dieses Problem effizient lösen zu können, gibt es die Möglichkeit mehrere Gitterebenen einzuführen, welches in dieser Arbeit nicht näher beschrieben werden soll (siehe [BFM$^+$07] für mehr Informationen).

5.7.2. Lokalitätsfilter

Um eine kurze Weganfrage von einer lange Weganfrage unterscheiden zu können, wird auf eine Distanzfunktion

$$m : V \times V \mapsto \mathbb{R}_0^+$$

zurückgegriffen. Falls eine s-t-Anfrage größer als eine vorher definierte Konstante K ist, so ist der Weg weit genug voneinander entfernt. Eine Definition von m kann auf unterschiedliche Art erfolgen. So kann der intuitiv Ansatz sein, dass die euklidische Distanz zweier Punkte genommen wird. Ein weiterer Ansatz könnte die Graphdistanz sein. Andernfalls muss eine lokale Suche mit einem anderen Konzept (beispielsweise der Dijkstra Algorithmus oder Contraction Hierarchies, welches besonders gut mit dem Transitknotenkonzept harmoniert) durchgeführt werden.

5.7.3. Distanzanfrage

Für eine s-t-Distanzanfrage lässt sich der Prozess folgendermaßen zusammenfassen:

1. Falls m(s,t) < K führe lokalen Anfrage aus, andernfalls fahre fort.
2. Frage alle Zugangsknoten zu s und t ab. Frage ebenfalls die Distanzen von s respektive t zu ihren jeweiligen Zugangsknoten ab.

3. Berechne für alle Zugangsknoten u,v die Distanz $d(s,u) + d(u,v) + d(v,t) : u \in AN(s), v \in AN(t)$.
4. Das Ergebnis der Anfrage ist das Minimum der Lösungen aus Schritt 3.

Da jeder dieser Werte schon vorberechnet ist, ist die Lösung in $\mathcal{O}(|AN_s| * |AN_t|)$ verfügbar.

5.7.4. Weganfrage

Die Route für eine s-t-Anfrage ist im Gegensatz zu der Distanz nicht gespeichert. Um dennoch eine Route ausgeben zu können, wird ausgehend von s der Knoten u_1, für den $d(u_1,t) = d(s,t) - d(s,u_1)$ gilt, gesucht. Anschließend wird die Suche von u_1 zu u_2 fortgeführt. Das heißt, es wird entsprechend der nächsten Kante ausgehend von u_1 zu dem Knoten u_2, die auf der Route liegt, gefolgt. Dieser Prozess wird solange durchgeführt, bis das Ziel t erreicht wurde. Dieses Vorgehen ist insbesondere deshalb möglich, da für die benötigten Teilstrecken die Längen bekannt sind.
Wie jedoch schon beschrieben, wird die Berechnung spätestens wenn m(u,t)<K ist, auf den lokalen Suchalgorithmus zurückgreifen.
Falls die s-t-Anfrage mindestens 2K voneinander entfernt ist, kann dies vermieden werden. Es wird startend von s nur die Route bis zu einem Punkt, der K entfernt vom Ziel t ist, betrachtet. Umgekehrt wird genauso für t in Richtung s Verfahren. Die Route kann entsprechend zusammengesetzt werden. Andernfalls besteht nur die Möglichkeit der Routenfindung über eine lokale Suche.

6. Transitknotenkonzept angewandt auf den Wikipedia-Linkgraph

In diesem Kapitel wird versucht, dass Transitknotenkonzept auf den Wikipedia-Linkgraph anzuwenden. Für den Erfolg dieses Vorhabens ist es essentiell, dass eine ausreichend kleine Transitknotenmenge (im Bereich $\mathcal{O}(\sqrt{V})$) gefunden wird. Die Arbeit von Eisner und Funke [EF12] hat gezeigt, dass die Transitknotenkonstruktion aus der Arbeit von Bast et al. [BFM09] eine Lösung findet, die nahe der optimalen Lösung liegt. Da der Wikipedia Graph jedoch keine geographischen Informationen enthält, stellt dieses Vorgehen (siehe Abschnitt 5.7) keine Option dar.
Um trotzdem eine Lösung finden zu können, wird das Problems des Auffindens der Transitknotenmenge als das Hitting-Set-Problem formuliert.
Das Hitting-Set-Problem ist ein NP-vollständiges Problem aus der Mengentheorie. Ausgehend von einem Universum U und einer Menge von Teilmengen S, ist eine Teilmenge $H \subset U$ gesucht, sodass jede Menge in S mindestens ein Element aus H enthält. Zusätzlich ist gefordert, dass das Hitting Set H einen gegebenen Wert k nicht überschreitet.
Angewandt auf den Wikipedia-Linkgraph ist das Universum U die Menge aller Knoten. Die Menge S stellt lange kürzeste Wege, das heißt Wege mit einer Mindestdistanz d, dar. Die Distanzfunktion, welche die Distanz definiert und somit später für den Lokalitätsfilter entscheidend ist, wird folgendermaßen definiert:

- Die Distanz d von einem Knoten A zu B sei definiert als die Anzahl der hops, die für einen kürzesten Weg von A nach B nötig ist (auch Level in der Breitensuche genannt).

Die Lösung H ist eine Menge von Knoten, die in jedem langen kürzesten Weg mindestens einen Knoten enthält, was $H \cap S \neq \emptyset$ entspricht. Diese Menge definiert die Transitknotenmenge.
Das Vorgehen für das Transitknotenproblem kann demnach folgendermaßen zusammengefasst werden:

1. Lege Distanz d für lange kürzeste Wege fest. Anfangen mit Distanz kleiner d werden mit der lokalen Suche bearbeitet.

2. Spanne mit Hilfe der Breitensuche für einen noch nicht untersuchten Knoten v den Wegbaum bis zur Distanz d auf.

3. Für den Knoten v (Teil des Universums U), der im Wegbaum einen Knoten w der Distanz d hat, wird eine Transitknotenuntersuchung gestartet. Diese Untersuchung wird für jeden Knoten w der Distanz d zu v durchgeführt. Im Beispiel der Abbildung 6.1 bedeutet dies, dass vom Startknoten A jeder Knoten mit Level d (die Knoten auf dem Ring der Distanz d) untersucht werden.

4. Wähle eine minimale Menge von Transitknoten aus (entspricht dem Hitting Set H), sodass jeder kürzester Weg der Länge d (welches der Menge aus Teilmengen S entspricht, wobei ein kürzester Weg Eine der Teilmengen beschreibt) aus Schritt 3 mindestens einen Transitknoten enthält. Es bietet sich bei dieser Auswahl an, den Startknoten auszuwählen, da hiermit alle Routen startend von ebendiesem Knoten v durch das Hitting Set *erschlagen* werden.

5. Markiere den Knoten v als besucht und fahre solange fort bis jeder Knoten im Graph bearbeitet wurde.

Mit diesem Vorgehen wird sichergestellt, dass ein kürzester Weg mit der Mindestdistanz d über einen Transitknoten führt. Dies schließt längere kürzeste Wege mit ein, da jeder kürzester Weg der Länge größer d als Präfix aus einem kürzesten Weg der Länge d besteht.
Für einen gerichteten Graph muss diese Berechnung für eingehende und ausgehende Kanten durchgeführt werden.

6.1. Kanonische Route und isReachable Pattern

Während einer Breitensuche ist die triviale Möglichkeit einen kürzesten Weg von A nach B zu bestimmen, für jeden Knoten den Vorgängerknoten zu speichern. Die Konstruktion des Weges ist somit entsprechend ein Aufruf aller Vorgängerknoten startend vom Zielknoten B. Diese Methode ist effizient und benötigt nur minimalen zusätzlichen Speicherplatz. Für die Konstruktion der Transitknotenmenge ist es jedoch nicht optimal. So hat ein kürzester Weg (im Weiteren kanonische Route genannt) von A nach B keinen Transitknoten, müsste entsprechend ein Transitknoten ausgewählt werden. Die Tatsache, dass ein alternativer kürzester Weg von A nach B nicht untersucht wird, bedeutet, dass unter Umständen eine größere Transitknotenmenge als nötig ausgewählt wird.
Um dieses Problem zu lösen, wird für jeden Knoten eine Markierung gesetzt. Genauer bedeutet dies, dass direkt bei der Breitensuche für Knoten A untersucht wird, ob der aktuell betrachtete Knoten über einen Transitknoten erreichbar ist. Dies ist genau dann der Fall, wenn einer der beiden folgenden Bedingungen für eine Markierung zutrifft:

- Der betrachtete Knoten ist selbst ein Transitknoten.
- Der im Wegbaum vorherige Knoten (das heißt der Knoten, der auf der Route um 1 näher am Startknoten liegt) wurde als über einen Transitknoten erreichbar markiert.

Im Verlauf dieser Arbeit wird auf dieses Vorgehen als *isReachable Pattern* verwiesen.

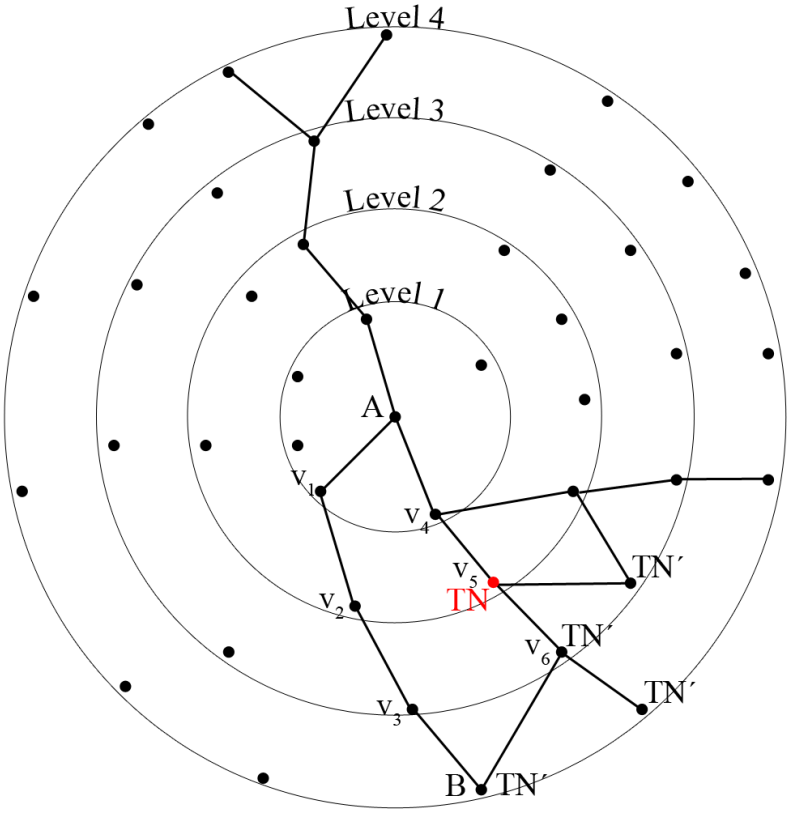

Abbildung 6.1.: Markierung der Knoten die über einen Transitknoten erreichbar sind. Der Übersichtlichkeit halber wurden nicht alle Wege eingezeichnet.

Am Beispiel von Abbildung 6.1 ist der rote Knoten v_5 ein Transitknoten. Alle im kürzesten Wegbaum für A folgenden Knoten von v_5 sind somit über einen Transitknoten erreichbar, weshalb diese Knoten mit TN' markiert sind.
Angenommen der kanonische Weg von A nach B ginge über $A\ v_1\ v_2\ v_3\ B$, so müsste für dieses Set ein Transitknoten gesetzt werden, da nach der Definition in Kapitel 6 dieses Set sonst nicht mit dem Hitting Set *erschlagen* wäre. Mit Hilfe der isReachable Markierung wird der Zielknoten B als über einen Transitknoten erreichbar markiert (für dieses Beispiel wäre dies über die Alternativroute $A\ v_4\ v_5\ v_6\ B$), sodass für dieses Set, welches das Ziel B hat, keine zusätzlichen Transitknoten ausgewählt werden müssen.

Mit diesem Vorgehen wird die potentielle Transitknotenmenge optimiert, da hierbei für den Startknoten auch alle Alternativwege in Betracht gezogen werden.

6.2. Konstruktion über zufällige Auswahl

Die trivialste Möglichkeit, eine Transitknotenmenge zu definieren, besteht aus einer rein randomisierten Auswahl der Knoten. Das heißt, dass Knoten zufällig ausgewählt werden, und auf diesen die

Transitknotenuntersuchung aus Kapitel 6 ausgeführt wird. Das Ergebnis dieser Untersuchung kann in Tabelle 6.1 für den EN Graph und in Tabelle 6.2 für den SIM Graph betrachtet werden, bei der die Transitknotenmenge und die benötigte Vorberechnungszeit auf Basis eines Threads jeweils für die kanonische Route und für das isReachable Pattern eingetragen sind.
In Folge der Randomisierung gilt zu berücksichtigen, dass konsekutive Untersuchungen auf die selbe Knotenmenge unterschiedliche Ergebnisse liefern können.

	kanonische Route		isReachable Pattern	
Distanz	# Transitknoten	Zeit [h]	# Transitknoten	Zeit [h]
2	6.534.585	1,39	6.523.038	1,79
3	6.476.982	92,99	6.426.256	152,70

Tabelle 6.1.: Transitknotenbestimmung über randomisierte Kontenuntersuchung (EN)

	kanonische Route		isReachable Pattern	
Distanz	# Transitknoten	Zeit [h]	# Transitknoten	Zeit [h]
2	97.823	0,01	96.373	0,01
3	95.032	0,05	92.646	0,09
4	93.206	0,36	91.886	0,68
5	89.658	0,84	84.723	1,98

Tabelle 6.2.: Transitknotenbestimmung über randomisierte Kontenuntersuchung (SIM)

So fällt für die zufällige Auswahl auf, dass die Transitknotenmenge weit von dem entfernt liegt, was idealerweise erreicht werden sollte. Zur Erinnerung: Diese sollte im Bereich $\mathcal{O}(\sqrt{V})$) liegen, was bei 9,6 Million Knoten und der erreichten Auswahl der Transitknotenmenge für Wege der Distanz 2 mit 6,5 Millionen und bei Wegen der Distanz 3 mit 6,4 Millionen Knoten verfehlt wurde. Für Distanzen größer 3 würde ein einzelner Thread laut Hochrechnung auf Basis der ersten 2,5 Millionen Knoten für eine erfolgreiche Berechnung über 50 Tage brauchen, weshalb diese hier nicht weiter aufgeführt werden.
Für den SIM Graph ergibt sich relativ gesehen ein ähnliches Bild. So ist die Transitknotenmenge für eine Ausgangsknotenmenge von 126.201 selbst bei Wegen der Länge 5 immer noch sehr hoch. Es fällt jedoch auf, dass das isReachable Pattern nach Tabelle 6.2 eine bis zu 6% bessere Auswahl trifft. Bei weiterführenden Berechnungen hat sich gezeigt, dass das isReachable Pattern bei Wegen bestimmter Länge eine um bis zu 50% bessere Auswahl trifft.
Die gemessenen Zeiten wurden auf Basis eines einzelnen arbeitenden Threads gemessen. Es fällt hierbei auf, dass die Zeiten sich zwischen den 2 Methoden zur Transitknotenauswahl ja nach Weglänge stark unterscheiden können. Dies ist mit der weiterreichenden Untersuchung des isReachable Pattern zu begründen, da hierbei zusätzlich zu der normalen Bearbeitung wie bei der kanonischen Route die Knoten auf Erreichbarkeit über Transitknoten untersucht werden. Je länger und zahlreicher die Wege werden, desto größer ist somit der zusätzliche Aufwand.
So ist das Fazit bei dieser Konstruktion, dass die Knotenmenge für das Transitknotenkonzept nicht

stark genug reduziert werden konnte. Im weiteren Verlauf wird versucht, die Transitknotenkonstruktion mit Hilfe bestimmter Methoden zu optimieren, um damit eine verringerte Menge aus Transitknoten bestimmen zu können.

6.3. Konstruktion über bevorzugte Auswahl von Knoten mit hoher Knotenvalenz

Bei dieser Variante wird versucht, eine geeignete Vorauswahl der Knoten zu berechnen. Hierbei werden die Knoten anhand ihrer Valenz geordnet, und entsprechend dieser Ordnung nach die Transitknotenuntersuchung ausgeführt.
Die Implementierung erfolgt hierbei über eine Prioritätswarteschlange, bei der die Priorität den Knotengrad widerspiegelt. Nach dem initialen Auffüllen der Warteschlange wird die Transitknotenuntersuchung für den ersten Knoten gestartet. Falls für den untersuchten Knoten w ein Transitknoten ausgewählt werden muss, so werden ebenso für alle Knoten, die w als Nachbarn haben, die Priorität angepasst. Dies bedeutet für Knoten ursprünglicher Priorität 1 ein Entfernen aus der Warteschlange und für benachbarte Knoten mit einer größeren Priorität eine Verringerung um Eins. Danach wird ein neuer Knoten aus der Warteschlange geholt, und solange weiter verfahren, bis die Warteschlange keine Knoten mehr enthält.
Im Folgenden sind die Ergebnisse dieser Untersuchung für den EN und SIM Graphen gelistet.

	kanonische Route		isReachable Pattern	
Distanz	# Transitknoten	Zeit [h]	# Transitknoten	Zeit [h]
2	3.546.154	1,70	3.542.694	2,16
3	2.981.289	108,36	2.958.784	154,89

Tabelle 6.3.: Transitknotenbestimmung über bevorzugte Untersuchung von Knoten mit hoher Valenz (EN)

	kanonische Route		isReachable Pattern	
Distanz	# Transitknoten	Zeit [h]	# Transitknoten	Zeit [h]
2	67.775	0,01	67.489	0,01
3	60.386	0,05	58.772	0,07
4	53.601	0,25	51.487	0,62
5	46.990	0,64	44.099	1,56

Tabelle 6.4.: Transitknotenbestimmung über bevorzugte Untersuchung von Knoten mit hoher Valenz (SIM)

Im Vergleich zu Abschnitt 6.2 liefert diese Methode eine um nahezu 50% kleinere Transitknotenmenge. Bei ausreichend großer Distanz erlaubt die beschriebene Optimierung und Neuordnung der Knoten

in der Warteschlange eine Beschleunigung der Vorberechnungsphase, da für aus der Warteschlange rausfallende Knoten kein kürzester Wegbaum aufgebaut werden muss.
Das Fazit bei dieser Konstruktion ist ähnlich wie in Abschnitt 6.2. Diese Methode liefert zwar in der Summe eine stark verbesserte Auswahl an Transitknoten, ist im Endeffekt jedoch immer noch nicht brauchbar, da das Ziel einer ausreichend kleinen Transitknotenmenge auch hier nicht erreicht werden konnte.

6.4. Konstruktion über Vorauswahl der Knoten durch das Vertex Cover

Mit den bis hierhin beschriebenen Konstruktionen wurde die potentielle Transitknotenmenge aus der kompletten Menge der Knoten bestimmt. In diesem Abschnitt wird versucht, die Auswahlmenge zu minimieren. Dies wird mit dem Vertex Cover erreicht.
Das Vertex Cover (auf Deutsch auch Knotenüberdeckungsproblem genannt) ist eines der klassischen NP-vollständigen Probleme. Für einen gegebenen Graph G ist eine Menge C von Knoten genau dann ein Vertex Cover, wenn alle Knoten in C jede Kante im Graph G *überdeckt*. Die triviale Lösung besteht darin, in das Set C einfach alle Knoten des Graphen G zu nehmen, weshalb zusätzlich eine Schranke k für die Größe von C gegeben ist.

Das Optimierungsproblem besteht in der Lösung des Minimalen Vertex Covers, das heißt, dass für den Graph G eine möglichst kleine Anzahl k an Knoten gefunden wird, sodass C einen Vertex Cover der Größe k beschreibt. Das Entscheidungsproblem beschreibt die Fragestellung, ob für einen Graph G und eine gegebene Schranke k der Graph G einen Vertex Cover der Größe höchstens k enthält.

Schlussendlich werden somit nur Transitknoten ausgewählt, die auch im Vertex Cover sind. Unter der Voraussetzung, dass Transitknoten so nah wie möglich am Startknoten ausgewählt werden sollen, hat durch die Definition des Vertex Covers dies zur Folge, dass explizit nur Startknoten oder Nachbarn von Startknoten in der Breitensuche ausgewählt werden können, da mindestens einer dieser Knoten im Vertex Cover enthalten ist.
Für das NP-vollständige Problem werden im Folgenden 2 Algorithmen vorgestellt, die das Vertex Cover Problem annäherungsweise lösen. Auf die Ergebnisse wird noch ein Optimierungsalgorithmus vorgestellt, der das Vertex Cover verkleinert.

6.4.1. 2-APX Algorithmus

Der 2-APX Algorithmus von Gavril [GJ79] beschreibt einen Algorithmus, der eine Lösung liefert, welches maximal um den Faktor 2 vom optimalen Ergebnis abweicht, d.h. es gilt $|C| \leq 2 * |C_{opt}|$.

Algorithmus 6.1 Vertex Cover 2-APX

$C \leftarrow \emptyset$
while $E \neq \emptyset$ **do**
  Wähle beliebige Kante $e = \{v, w\} \in E$
  $C \leftarrow C \cup \{v, w\}$
  Lösche alle adjazenten Kanten von v, w
end while
return C

Der Algorithmus 6.1 wählt eine beliebige Kante $e = \{v, w\} \in E$ aus, fügt beide Knoten zu C hinzu, und löscht alle adjazenten Kanten von v und w. Dieser Schritt wird solange wiederhole, bis alle Kanten aus dem Graph gelöscht worden sind.
Mit diesem Vorgehen wird eine disjunkte Menge M an Kanten ausgewählt, wodurch ein Cover C der Größe $2 * |M|$ produziert wird. Die optimale Lösung C_{opt} hat mindestens die Größe $|M|$, wodurch sich $|C| = 2 * |M| \leq 2 * C_{opt}$ ergibt.

Der EN und der SIM Graph haben mit diesem Approximationsalgorithmus folgende Werte:

EN Graph	SIM Graph
5.323.043	87.695

Tabelle 6.5.: Vertex Cover Größe mit 2-APX

Damit wird die Menge der Knoten, aus denen ein Transitknoten ausgewählt wird, um 45% für den EN Graph und um 31% für den SIM Graph, verkleinert.

6.4.2. Greedy Algorithmus

Eine weitere Möglichkeit einen Vertex Cover zu bestimmen, sei der im folgenden beschriebene Algorithmus.

1. Wähle den Knoten v mit maximalem Grad aus und füge diesen in das Vertex Cover hinzu.
2. Entferne v sowie alle adjazenten Kanten zu v.
3. Aktualisiere die Warteschlange (Priorität von Knoten, die mit rausfallenden Kanten verbunden sind, muss angepasst werden).
4. Wiederhole die Schritte 1. und 2. bis G keine Kanten mehr enthält.

Algorithmus 6.2 Vertex Cover Greedy

```
C ← ∅
Fülle die Warteschlange PQ mit allen Knoten und den Knotengraden als Priorität.
while PQ ≠ ∅ do
    Wähle Knoten v ∈ PQ mit höchster Priorität
    C ← C ∪ {v}
    Aktualisiere_PQ(v)
end while
return C
```

Die Implementierung in Algorithmus 6.2 erfolgt hierbei mit einer Prioritätswarteschlange, bei der ein Knotengrad die Priorität darstellt. Das Entfernen der Kanten wird durch eine Markierung erreicht, das heißt, dass die Kanten nicht wirklich entfernt werden sondern nur als entfernt markiert worden sind. Mit Hilfe dieser Markierung wird bei jeder Aktualisierung der Warteschlange überprüft, welche

Knoten mit dem Knoten v verbunden sind, und ihr Knotengrad wird bei Vorhandensein einer Kante mit v um eins erniedrigt bzw. aus der Warteschlange entfernt falls die Priorität o erreicht.

EN Graph	SIM Graph
3.411.041	60.384

Tabelle 6.6.: Vertex Cover Größe mit Greedy

Die Tabelle 6.6 zeigt, dass das Resultat eine um 30-35% geringere und damit bessere Auswahl an Knoten als der 2-APX Algorithmus trifft.
Dieser Algorithmus scheint somit intuitiv und auch durch die Auswertung gezeigt, einen besseren Vertex Cover zu bestimmen. Dieser Schein trügt jedoch, da der mathematische Beweis zeigt, dass dieser Algorithmus die optimale Lösung im worst case Fall nicht besser als Faktor $logn$ approximiert. Dessen ungeachtet, kommt dieser Algorithmus in der Praxis, wie in Tabelle 6.6 gezeigt, im Allgemeinen auf bessere Werte.

6.4.3. Optimierung

Für die beiden beschriebenen Algorithmen gibt es zusätzlich noch die Möglichkeit, die ausgewählte Menge C aus Unterabschnitt 6.4.1 und Unterabschnitt 6.4.2 zu optimieren. Diese Optimierung, die eine worst case Laufzeit von $\mathcal{O}(|E| + |V|)$ hat, arbeitet auf einer Vertex Cover Eingabemenge C wie folgt:

- Wähle einen noch unbearbeiteten Knoten $v \in C$ aus.
- Untersuche, ob jeder Nachbar u zu v ebenfalls in C liegt. Falls dies zutrifft, wird der Knoten v aus der Menge C entfernt.
- Wiederhole solange, bis jeder Knoten in C so abgearbeitet wurde.

Das Ergebnis dieser Optimierung kann in Tabelle 6.7 betrachtet werden.

	EN Graph		SIM Graph	
	Anzahl	Optimierte Knoten	Anzahl	Optimierte Knoten
2-APX	3.454.221	1.868.822	61.501	26.194
Greedy	3.402.514	8.527	60.233	151

Tabelle 6.7.: Vertex Cover Größe nach Optimierung

Es fällt auf, dass die Optimierung unterschiedlich stark ausfällt. So wird das Vertex Cover, welches auf Basis des 2-APX berechnet wurde, mit über 1.8 Millionen bzw. 26.194 wegfallenden Knoten stark verkleinert. Das auf dem Greedy Algorithmus basierende Vertex Cover wird durch diesen Prozess mit 8.527 bzw. 151 Knoten nur minimal optimiert, welches auf eine bessere initiale Auswahl der Knoten

in Unterabschnitt 6.4.2 schließen lässt. Das optimierte Vertex Cover unterscheidet sich nur minimal zwischen den beiden Algorithmen, wobei für alle untersuchten Graphen (inklusive der Auswertungen der Graphen im Anhang A) der Greedy Algorithmus eine bessere initiale und optimierte Auswahl geliefert hat.

6.4.4. Konstruktion der Transitknotenmenge

Bei der Konstruktion über das Vertex Cover wird größtenteils wie in Kapitel 6 vorgegangen. Der einzige Unterschied hierbei ist, dass bei der Auswahl eines Transitknotens darauf geachtet werden muss, dass dieser Knoten ebenfalls im Vertex Cover vertreten ist. Dies heißt in der Praxis, dass entweder der Startknoten oder ein Nachbar des Startknotens ausgewählt wird. Durch diese Vorauswahl der Knoten wird die Knotenanzahl, aus der der Transitknotenalgorithmus auswählen kann, mit der Hoffnung reduziert, dass die dadurch entstehende Transitknotenmenge ausreichend klein wird.

	kanonische Route		isReachable Pattern	
Distanz	# Transitknoten	Zeit [h]	# Transitknoten	Zeit [h]
2	2.938.656	0,71	2.929.772	1,60
3	2.548.324	46,09	2.522.734	86,43

Tabelle 6.8.: Transitknotenbestimmung über vorherige VC Auswahl (EN)

Das Ergebnis dieser Untersuchung in Tabelle 6.8 zeigt für den EN Graph, dass diese Methode die bisher kleinste Menge an Transitknoten liefert. Mit ca. 2,5 Millionen Knoten für Wege der Distanz 3 bedeutet dies eine geringere Menge als die Auswahl über den Knotengrad und ebenfalls eine geringere Menge als die zufällige Auswahl.

	kanonische Route		isReachable Pattern	
Distanz	# Transitknoten	Zeit [h]	# Transitknoten	Zeit [h]
2	48.618	0,01	48.149	0,01
3	41.240	0,04	40.799	0,06
4	35.765	0,25	34.875	0,52
5	31.346	0,66	30.049	1.68

Tabelle 6.9.: Transitknotenbestimmung über vorherige VC Auswahl (SIM)

Die Untersuchung auf den SIM Graph liefert ein relativ ähnliches Bild. Die Anzahl der Transitknoten ist teilweise stark kleiner (teilweise um ein Drittel geringer) und ist somit auch hier minimal im Vergleich zu den anderen Konstruktionen.
Bei allen Konstruktionen liefert das isReachable Pattern ein bessere Ergebnis, welches jedoch mit einer längeren Berechnungsphase in Kauf genommen werden muss.

Summa summarum ist das Ergebnis jedoch ernüchternd. So ist die Anfangs erhoffte Reduktion der Transitknotenmenge in den Bereich von $\mathcal{O}(\sqrt{V})$ fehlgeschlagen. Eine Kantenminderung wie in Abschnitt 3.4 beschrieben, kommt durch die ohnehin große Transitknotenmenge und die beschränkte Anzahl der zu kontrahierenden Kanten auch nicht in Frage. Die niedrigsten gefundenen Transitknotenmengen sind immer noch so hoch, wodurch das Anwenden des Transitknotenkonzeptes nicht effektiv durchführbar wird.

6.5. Untere Schranke disjunkter Wege

Um die vorangegangenen Berechnungen zu bestätigen und auch einen Beweis zur Nichtdurchführbarkeit zu erhalten, wird dieses Unterkapitel versuchen, eine untere Schranke von Knoten zu bestimmen, zu der Knoten mit bestimmter Distanz mindestens eine untere Grenze an Transitknoten enthalten müssen. Die Überlegung ist wie folgt und gilt nur für einen bestimmten Weg (es ist daher keine Schranke für das isReachable Pattern, bei der auch Alternativrouten betrachtet werden):

- Finde eine Menge an Wegen mit Distanz d, die vollständig disjunkt zueinander sind.
- Die Anzahl dieser Wege ist somit eine untere Schranke, da jeder dieser Wege nach Definition in Kapitel 6 einen Transitknoten setzen müsste.

Die Implementierung besteht hierbei im Grunde aus dem Schritt 1 und 2 in der folgenden Beschreibung:

1. Spanne mit der Breitensuche für einen noch nicht betrachteten Knoten v den Wegbaum bis zur Distanz d auf.
2. Für alle Knoten w mit Distanz d zu v, suche einen kürzesten Weg, der aus Knoten besteht, die bisher in keinem anderen Weg genutzt worden sind. Dies erfolgt mit Hilfe einer modifizierten rückwärts gerichteten Tiefensuche, der startend von w nur Knoten entlang geht, die nicht markiert sind (= disjunkt zu anderen bereits gefundenen Wegen) und die einen Level näher am Startknoten liegen (nutzt die berechneten Distanzen aus Schritt 1). Falls eine Tiefensuche so am Startknoten ankommt, wurde ein Weg gefunden und die weitere Bearbeitung der Knoten w kann abgebrochen werden.
3. Falls ein solcher Weg gefunden wurde, markiere die genutzten Knoten und erhöhe die untere Schranke.
4. Fahre mit dem nächsten Knoten fort, bis alle Knoten $v \in V$ untersucht worden sind.

Die Tabelle 6.10 zeigt das Ergebnis für den EN als auch für den SIM Graph.

	EN Graph		SIM Graph	
Distanz	Schranke	Zeit [h]	Schranke	Zeit [h]
2	1.448.799	0,77	22.282	0,01
3	866.960	73	13.625	0,06
4	-	-	9.193	0,60
5	-	-	6.647	2,39

Tabelle 6.10.: Untere Schranke für disjunkte kürzeste Wege

Der EN Graph hat für kürzeste Wege der Länge 2 insgesamt über 1,4 Millionen Wege, die aus vollständig disjunkten Knoten bestehen. Für kürzeste Wege der Länge 3 liegt diese Anzahl immer noch bei über 800.000. Untersuchungen für längere Wege hätten für den EN Graph den Zeitrahmen gesprengt, weshalb Untersuchungen für kürzeste Wege der Länge 4 und 5 nur für den SIM Graph erstellt wurden.

Das Ergebnis zeigt, dass die vorgestellten unterschiedlichen Algorithmen für die Konstruktion der Transitknotenmenge, vor allem Abschnitt 6.4, eine gute Näherung finden, jedoch kein besseres Ergebnis als die gezeigte untere Schranke für die jeweilige Distanz liefern können.
Zu beachten gilt hierbei, dass dies eine untere Schranke ist und eine Auswahl eines einzelnen Transitknotens für einen disjunkten Weg nicht zwingend alle Wege in der Hitting Set Untersuchung erschlägt. Falls Transitknoten anhand dieser Untersuchung ausgewählt werden sollten, so müsste jeder Knoten, der Bestandteil eines disjunkten kürzesten Wegs ist, in die Transitknotenmenge hinzugefügt werden (zur Erinnerung: Distanzen der Länge d haben insgesamt $d + 1$ Knoten).
Dies erklärt auch, wieso die initiale Zielsetzung des Auffindens einer Transitknotenmenge im Bereich $\mathcal{O}(\sqrt{V})$ verfehlt wurde, denn auch für kürzeste Wege der Distanz 5 im SIM Graph ist mit 6.647 disjunkten Wegen ein Erreichen dieser Menge nicht möglich. Somit wurde gezeigt, weshalb die Transitknotenkonstruktionen ihr Ziel verfehlt haben.

Eine entsprechende Untersuchung für eine untere Schranke, bei der alle Wege (inklusiver Alternativwege) betrachtet werden, müsste ein Set anstatt aus einzelnen Wegen der Distanz d aus einer Menge von Knoten bestehen, über die auch jeder Alternativweg führt. Dies impliziert, dass die Sets, die in der vorherigen Untersuchung eine bestimmte Größe hatten, mit dieser Betrachtung sehr viel größer werden. Dadurch sollte die Anzahl der Sets geringer ausfallen, da auch hier diese Sets (entspricht kürzesten Wegen von einem Startknoten zu einem Zielknoten inklusiver aller Alternativwege) vollständig disjunkt sein müssen.

In Anlehnung an die ursprüngliche Implementierung aus Abschnitt 6.5 könnte hierbei folgendermaßen vorgegangen werden:

1. Spanne mit der Breitensuche für einen noch nicht betrachteten Knoten v den Wegbaum bis zur Distanz d auf.
2. Für alle Knoten w mit Distanz d zu v, suche alle kürzesten Weg von v zu w (beispielsweise mit einer modifizierten Breitensuche, welche die Informationen aus 1. nutzt).
3. Falls jeder Knoten in einer solchen Knotenmenge noch nicht von einer anderen Knotenmenge genutzt worden ist, wurde eine neue disjunkte Menge gefunden. Markiere die Knoten aus der Menge und erhöhe die untere Schranke.
4. Fahre mit dem nächsten Knoten fort, bis alle Knoten $v \in V$ untersucht worden sind.

7. Zusammenfassung und Ausblick

In diesem Kapitel wird eine Zusammenfassung dieser Arbeit präsentiert und Vorschläge für Anknüpfungspunkte für die Weiterentwicklung diskutiert.

7.1. Zusammenfassung

Im Rahmen dieser Diplomarbeit wurde die Transitknoten Beschleunigungstechnik auf den Wikipedia-Linkgraph angewandt.

Für den Erfolg dieses Unterfangens wurde nach der Extraktion des Linkgraphen die Struktur des Graphen untersucht. Um eine Vergleichsbasis mit anderen Graphen zu erhalten, wurde im Text zur Wikipedia in englischer Sprache und auf Simple English, einer vereinfachten Form der englischen Sprache, Bezug genommen. Im Anhang wurden die Resultate für die deutsche und französische Wikipedia ergänzt.
Um für die darauffolgende Implementierung des Beschleunigungsalgorithmus eine Vergleichsgrundlage zu erhalten, wurde die Breitensuche implementiert und Benchmarks durchgeführt. Nach einer einleitenden Vorstellung diverser Beschleunigungstechniken wurde mit verschiedenen Methoden versucht, das Transitknotenkonzept zu implementieren. Jedoch konnte keiner dieser Methoden eine ausreichend kleine Transitknotenmenge (im Bereich von $\mathcal{O}(\sqrt{V})$) erzeugen.
Anschließend wurde über eine untere Schranke, welche aus vollständig disjunkten kürzesten Wegen bestimmter Länge besteht, gezeigt, dass die implementierten Methoden gute Näherungen an diese Schranke liefern. Jedoch zeigt diese untere Schranke auch, dass die eingangs geforderte Berechnung einer kleinen Transitknotenmenge nicht realisierbar ist, wodurch ein Anwenden des Transitknotenkonzeptes auf den Wikipedia-Linkgraph fehlschlägt.

7.2. Ausblick

Der Ansatz dieser Arbeit hat in unterschiedlicher Hinsicht Verbesserungspotential, welche in diesem Abschnitt erwähnt werden und Anknüpfungspunkte und Weiterentwicklungsmöglichkeiten aufzeigen sollen.

Daten ohne Weiterleitungsartikel

Diese Arbeit basiert auf allen Artikeln im Wikipedia Namensraum 0. Dies heißt, dass auch Weiterleitungsartikel (für unterschiedliche Schreibweisen, Abkürzungen und Ähnliches) in der Graphstruktur enthalten sind. Diese Artikel bestehen aus einer direkten Weiterleitung zum Hauptartikel.
Eine erneute Untersuchung der Graphen, jedoch ohne die erwähnten Weiterleitungsartikel, sollte die Graphstruktur vereinfachen und könnte eventuell vorhandene Differenzen aufzeigen.

Gesamte Wikipedia als Datengrundlage

Mit mindestens 40 Millionen unterschiedlichen Ids ist die gesamte Wikipedia mit allen Artikeln in jedem Namensraum sehr viel größer als der in dieser Arbeit untersuchte Graph.
Die Auswertung für diesen Graph könnte durch die differente Linkstruktur unterschiedliche Ergebnisse liefern und somit diese Arbeit erweitern.

Webservice

Die Implementierung als Webservice sollte auf Grundlage dieser Arbeit kein großes Hindernis darstellen. Für einen Nutzer ist es sicherlich interessant zu sehen, wie und über welche Artikel ein Weg von einem Startartikel zu einem Zielartikel führt. Bei der Implementierung sollte darauf geachtet werden, dass dem Nutzer die Möglichkeit einer gerichteten und ungerichteten Suche gegeben wird.

Parallelisierung optimieren

Einige Implementierungsabschnitte in dieser Arbeit nutzen zwar eine Parallelisierung, jedoch stellt die starke gemeinsame Nutzung des Arbeitsspeichers unterschiedlicher Prozessorkerne einen Flaschenhals dar. Wie diese Realisierung, beispielsweise mit einem Datenbankserver, erfolgen soll, muss untersucht werden.
Ebenfalls sollten die nicht parallelisierten Berechnungen entsprechend parallelisiert werden, was im Optimalfall eine Beschleunigung mit sich bringt.

Auswertung weiterer Beschleunigungstechniken

Wie in Kapitel 5 gezeigt wurde, gibt es die Möglichkeit, andere Beschleunigungstechniken zu verwenden. Eine entsprechende Arbeit, welche die Graphextraktion und die Analyse des Graphen aus dieser Arbeit als Grundlage verwendet, könnte untersuchen, ob eine der erwähnten Beschleunigungstechniken ein besseres Resultat erzielt.

A. Appendix

CPU	Intel Core i7-3930K @ 3.2GHz
Arbeitsspeicher	64GB DDR3
Festplatte	Seagate Barracuda 7200.12 ST31000528AS
Betriebssystem	Ubuntu 3.2.0-29-generic 64bit
Java Version	1.6.0_24
JVM	OpenJDK RE (IcedTea6 1.11.5)
JVM-Optionen	-Xmx8g -Xms8g -Xss1g
Bibliotheken	Stanford CoreNLP version 1.3.4

Tabelle A.1.: Leistungsdaten und JVM Optionen des zugrundeliegenden Systems

University_of_Stuttgart → Berlin_Institute_of_Technology → Moon →
List_of_minor_planets → List_of_minor_planets:_100001-101000 →
Brian_G._W._Manning → List_of_asteroids/100601-100700 →
List_of_minor_planets/100601-100700 → ... → List_of_minor_planets/191701-191800 →
List_of_minor_planets/191801-191900 → List_of_minor_planets/191901-192000

Abbildung A.1.: Beispielstruktur eines Weges im EN Graph mit Länge 1.361

Graph Details	DE	FR
Knoten	2.576.857	2.594.730
Kanten	58.117.467	72.990.950
Knoten ohne eingehende Kanten	756.049	797.090
Knoten ohne ausgehende Kanten	7.972	1.757
∅ Kanten pro Knoten	22,55	28,13
Längster kürzester Weg	16	16

Tabelle A.2.: Allgemeine Informationen zum Linkgraph (DE und FR)

Grad	Für ausgehende Kanten		Für eingehende Kanten	
	Bereich	Gesamtanzahl	Bereich	Gesamtanzahl
0	7.972	7.972 (0,31%)	756.049	756.049 (29,34%)
1	1.076.750	1.076.750 (41,79%)	262.344	262.344 (10,18%)
2 - 20	29.947 - 54.745	678.414 (26,33%)	16.469 - 162.507	1.099.995 (42,69%)
21 - 40	12.409 - 33.044	430.367 (16,7%)	5.745 - 15.206	193.768 (7,52%)
41 - 60	5.298 - 12.005	158.881 (6,17%)	3.126 - 5.710	85.245 (3,31%)
61 - 80	2.807 - 4.983	76.819 (2,98%)	1.867 - 2.964	48.812 (1,89%)
81 - 100	1.624 - 2.683	44.007 (1,71%)	1.139 - 2.030	31.277 (1,21%)
101 - 200	206 - 1.673	73.310 (2,84%)	222 - 1.412	60.198 (2,34%)
201 - 300	86 - 344	18.169 (0,71%)	58 - 434	17.655 (0,69%)
301 - 400	18 - 148	6.491 (0,25%)	34 - 318	8.415 (0,33%)
401 - 500	7 - 82	2.310 (0,09%)	17 - 203	3.750 (0,15%)
501 - 1000	0 - 36	2.850 (0,11%)	0 - 52	5.295 (0,21%)
> 1000	0 - 6	517 (0,02%)	0 - 26	4.054 (0,16%)

Tabelle A.3.: Untersuchung der Knotenvalenz (DE)

Grad	Für ausgehende Kanten		Für eingehende Kanten	
	Bereich	Gesamtanzahl	Bereich	Gesamtanzahl
0	1.757	1.757 (0,07%)	797.090	797.090 (30,72%)
1	1.255.493	1.255.493 (48,39%)	280.577	280.577 (10,81%)
2 - 20	16.565 - 33.142	507.561 (19,56%)	14.338 - 199.982	1.021.338 (39,36%)
21 - 40	11.324 - 21.977	343.422 (13,24%)	5.636 - 13.296	178.454 (6,88%)
41 - 60	5.633 - 10.661	151.736 (5,85%)	3.485 - 5.479	88.324 (3,4%)
61 - 80	3.726 - 5.486	92.133 (3,55%)	2.195 - 3.658	54.845 (2,11%)
81 - 100	2.421 - 3.615	60.848 (2,35%)	1.496 - 2.337	37.498 (1,45%)
101 - 200	443 - 2.557	122.245 (4,71%)	340 - 1.721	80.688 (3,11%)
201 - 300	137 - 646	35.771 (1,38%)	124 - 607	26.971 (1,04%)
301 - 400	47 - 346	12.157 (0,47%)	39 - 364	10.815 (0,42%)
401 - 500	19 - 169	5.447 (0,21%)	21 - 110	5.073 (0,2%)
501 - 1000	0 - 133	5.741 (0,22%)	0 - 103	8.541 (0,33%)
> 1000	0 - 8	419 (0,02%)	0 - 29	4.516 (0,17%)

Tabelle A.4.: Untersuchung der Knotenvalenz (FR)

Titel	Anzahl ausgehender Kanten
Liste_von_Vornamen	5.212
Liste_von_Automobilmarken	4.145
Liste_der_französischen_Kantone	3.961
Liste_von_Begräbnisstätten_bekannter_Persönlichkeiten	3.927
Liste_der_katholischen_Diözesen	3.305
Liste_von_Kriegsfilmen	3.283
Liste_von_Ortsteilen_in_Hessen	3.029
Jahreskalender	2.974
Abkürzungen/Gesetze_und_Recht	2.839
Liste_von_Erfindern	2.791

Tabelle A.5.: Auflistung der Knoten geordnet nach Anzahl ausgehender Kanten (DE)

Titel	Anzahl ausgehender Kanten
Liste_de_sigles_de_trois_lettres	7.514
Liste_des_bureaux_de_poste_français_classés_par _oblitération_Gros_Chiffres	5.323
Localités_de_Serbie	4.771
Liste_des_bureaux_de_poste_français_classés_par _oblitération_Petits_Chiffres	4.614
Liste_des_cantons_français	4.139
Liste_des_routes_nationales_de_France	3.813
Liste_des_villes_et_villages_fleuris_de_France	3.765
Liste_de_gares_de_France	3.423
Liste_des_communes_de_Suisse	2.908
Liste_des_codes_postaux_belges	2.775

Tabelle A.6.: Auflistung der Knoten geordnet nach Anzahl ausgehender Kanten (FR)

Titel	Anzahl eingehender Kanten
Virtual_International_Authority_File	198.735
Library_of_Congress_Control_Number	145.023
Gemeinsame_Normdatei	120.127
Personennamendatei	108.598
Höhe_über_dem_Meeresspiegel	105.513
Vereinigte_Staaten	102.016
Deutschland	97.385
Deutsche_Nationalbibliothek	87.997
Internet_Movie_Database	76.342
Frankreich	68.332

Tabelle A.7.: Auflistung der Knoten geordnet nach Anzahl eingehender Kanten (DE)

Titel	Anzahl eingehender Kanten
France	247.542
Liste_des_pays_du_monde	177.299
Coordonnées_géographiques	148.994
États-Unis	142.454
Superficie	107.650
Densité_de_population	105.259
Recensement_de_la_population	101.601
Altitude	91.160
Code_postal	87.191
2008	81.068

Tabelle A.8.: Auflistung der Knoten geordnet nach Anzahl eingehender Kanten (FR)

Komponentengröße	1	2	3	4	5	9	2.569.738
Vorkommen DE	5.320	830	20	15	2	1	1
Komponentengröße	1	2	3	4	5	6	2.593.749
Vorkommen FR	619	131	17	8	1	2	1

Tabelle A.9.: Auswertung schwacher Zusammenhang (DE und FR)

Komponentengröße	1	2	3	4	5	6-26	1.780.535
Vorkommen DE	788.586	2.792	426	97	22	39	1
Komponentengröße	1	2	3	4	5	6-17	1.780.329
Vorkommen FR	811.861	806	160	44	16	24	1

Tabelle A.10.: Auswertung starker Zusammenhang (DE und FR)

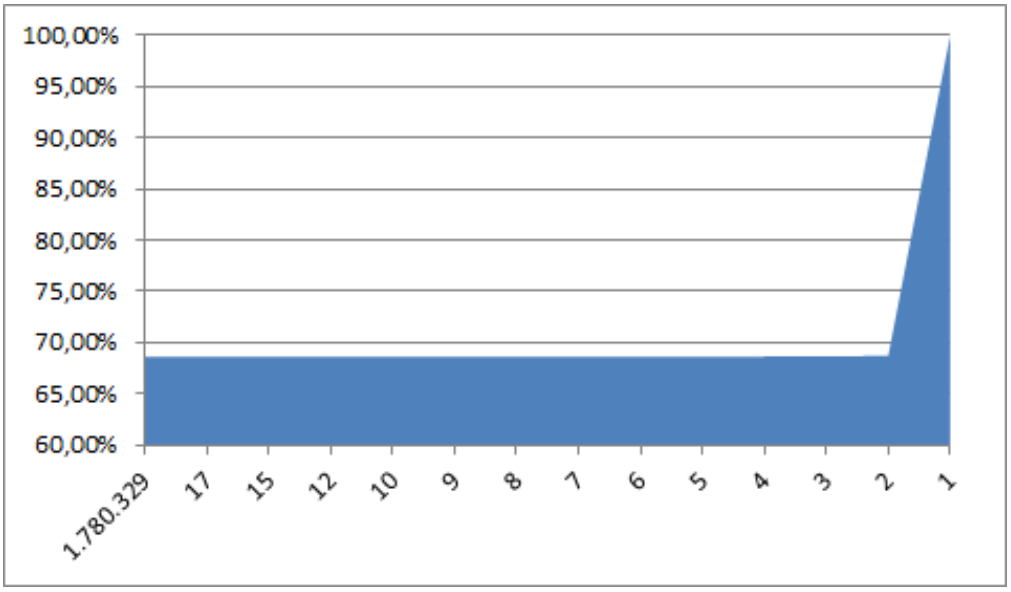

Abbildung A.2.: Starke Zusammenhangskomponenten (DE)
x-Achse: Komponentengröße
y-Achse: Anteil an Gesamtknotenmenge

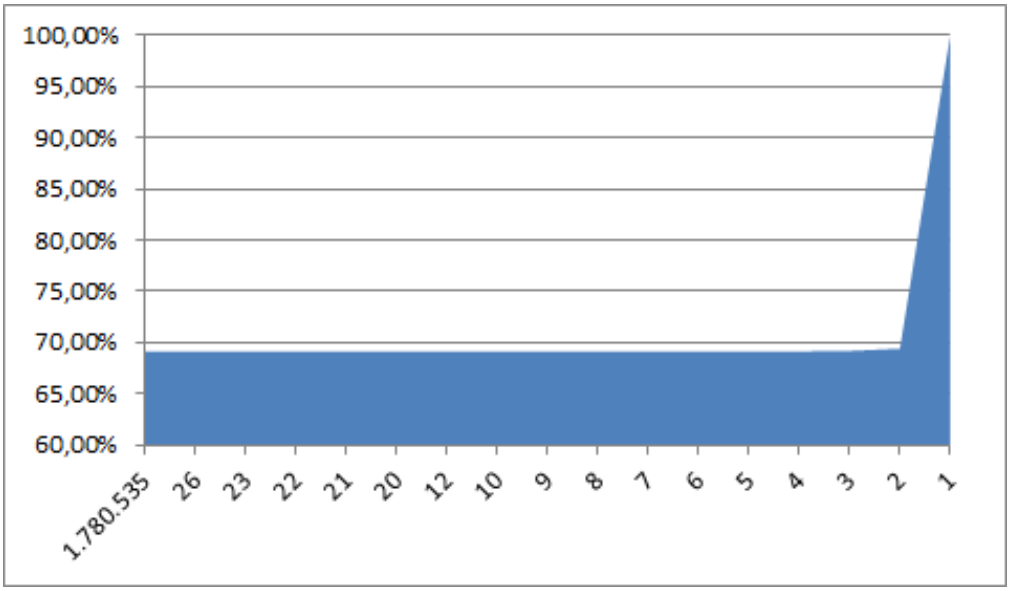

Abbildung A.3.: Starke Zusammenhangskomponenten (FR)
x-Achse: Komponentengröße
y-Achse: Anteil an Gesamtknotenmenge

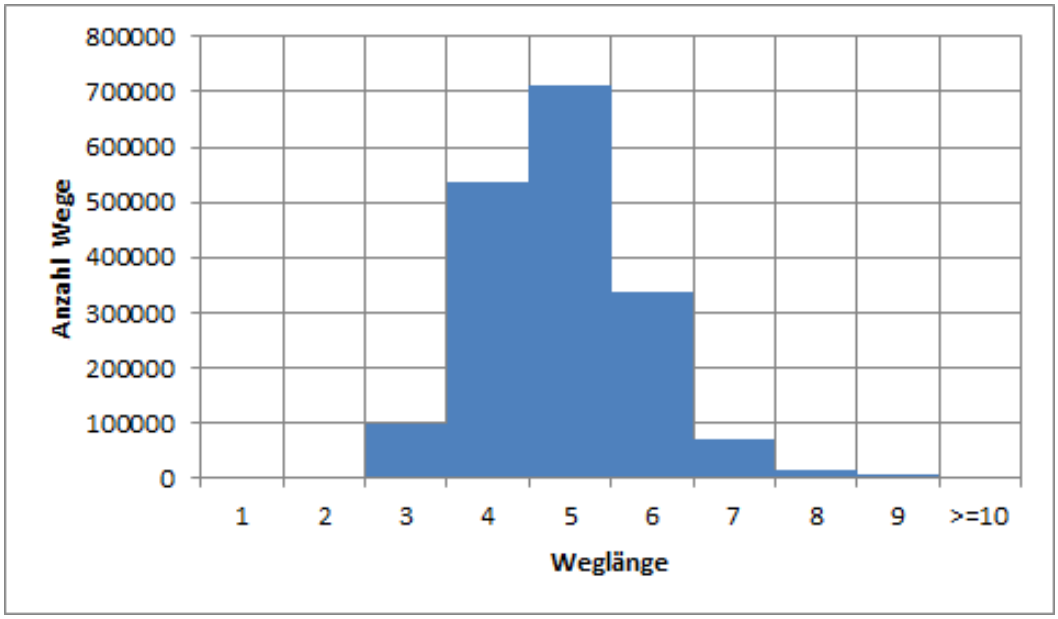

Abbildung A.4.: Anzahl der kürzesten Wege (DE)
(Stichprobengröße: 10.000)

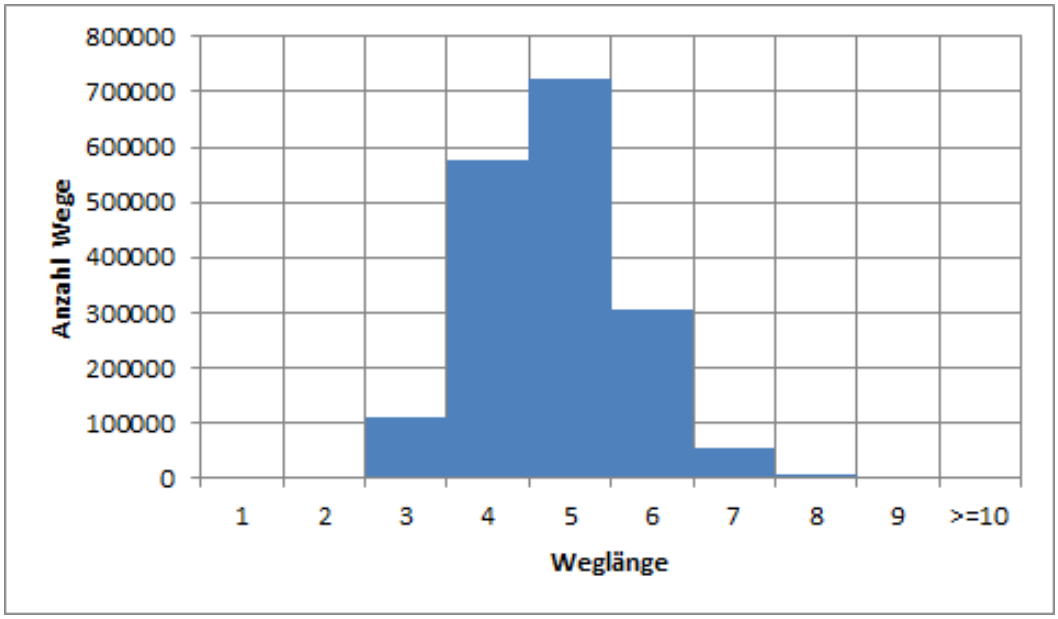

Abbildung A.5.: Anzahl der kürzesten Wege (FR)
(Stichprobengröße: 10.000)

Weglänge	2	3	4	5	6	7	8	≥9	∞
∅ Lfz [ms]	0,4	15	105	264	535	940	1150	1184	1209
Anteil	0,1%	4%	21%	27,4%	12,6%	2,7%	0,6%	0,2%	31,3%
(ohne Wege der Länge ∞)	0,1%	5,8%	30,6%	39,9%	18,4%	4%	0,9%	0,35%	-

Tabelle A.11.: Anfragezeiten zur Breitensuche (DE)

Weglänge	2	3	4	5	6	7	8	≥9	∞
∅ Lfz [ms]	0,4	16,7	112	293	589	1003	1167	1306	1310
Anteil	0,1%	4,2%	21,8%	27,7%	11,9%	2,2%	0,3%	0%	31,8%
(ohne Wege der Länge ∞)	0,1%	6,1%	32%	40,6%	17,5%	3,2%	0,4%	0,04%	-

Tabelle A.12.: Anfragezeiten zur Breitensuche (FR)

	DE			FR		
Länge der kürzesten Wege	2	3*	4**	2	3*	4**
Anzahl kürzester Wege	$5,7 \times 10^9$	255×10^9	$1,5 \times 10^{12}$	$7,1 \times 10^9$	280×10^9	$1,7 \times 10^{12}$
∅ Anzahl kürzester Wege pro Knoten	2.244	98.958	582.104	2.736	107.911	655.174
Zeit [h] inkl. Speichern (1 T)	0,68	41,1	340	0,86	43,5	390
Zeit [h] inkl. Speichern (6 T)	0,38	24,2	144	0,46	27,9	164
Zeit [h] ohne Speichern (1 T)	0,18	13,81	146	0,24	15,31	171
Zeit [h] ohne Speichern (6 T)	0,05	3,82	44	0,07	4,27	58
Platzverbrauch [GB]	125	7.600	54.000	154	7.900	59.770

Tabelle A.13.: Extraktion kürzester Wege
* Hochrechnung auf Basis von 100.000 Knoten
** Hochrechnung auf Basis von 10.000 Knoten

Distanz	kanonische Route		isReachable Pattern	
	# Transitknoten	Zeit [h]	# Transitknoten	Zeit [h]
2	1.928.404	0,29	1.919.153	0,39
3	1.901.799	16,43	1.888.315	23,34
4	1.869.988	140	1.822.193	192

Tabelle A.14.: Transitknotenbestimmung über randomisierte Kontenuntersuchung (DE)

Distanz	kanonische Route		isReachable Pattern	
	# Transitknoten	Zeit [h]	# Transitknoten	Zeit [h]
2	1.893.301	0,37	1.880.322	0,46
3	1.855.522	22,07	1.832.415	33,10
4	1.798.245	153	1.758.515	212

Tabelle A.15.: Transitknotenbestimmung über randomisierte Kontenuntersuchung (FR)

Distanz	kanonische Route		isReachable Pattern	
	# Transitknoten	Zeit [h]	# Transitknoten	Zeit [h]
2	1.235.855	0,32	1.232.791	0,34
3	1.083.786	22,81	1.072.193	28,56
4	979.113	180	971.158	301

Tabelle A.16.: Transitknotenbestimmung über bevorzugte Untersuchung von Knoten mit hoher Valenz (DE)

Distanz	kanonische Route		isReachable Pattern	
	# Transitknoten	Zeit [h]	# Transitknoten	Zeit [h]
2	1.132.321	0,39	1.128.463	0,49
3	989.600	25,49	979.476	32,24
4	897.078	190	879.852	350

Tabelle A.17.: Transitknotenbestimmung über bevorzugte Untersuchung von Knoten mit hoher Valenz (FR)

DE Graph	FR Graph
1.791.204	1.635.280

Tabelle A.18.: Vertex Cover Größe mit 2-APX (DE und FR)

DE Graph	FR Graph
1.188.512	1.069.454

Tabelle A.19.: Vertex Cover Größe mit Greedy (DE und FR)

	DE Graph		FR Graph	
	Anzahl	Optimierte Knoten	Anzahl	Optimierte Knoten
2-APX	1.206.298	584.906	1.085.921	549.359
Greedy	1.184.782	3.730	1.066.588	2.866

Tabelle A.20.: Vertex Cover Größe nach Optimierung (DE und FR)

	kanonische Route		isReachable Pattern	
Distanz	# Transitknoten	Zeit [h]	# Transitknoten	Zeit [h]
2	1.025.627	0,13	1.022.554	0,23
3	893.846	10,04	882.983	15,89
4	808.072	113	789.840	230

Tabelle A.21.: Transitknotenbestimmung über vorherige VC Auswahl (DE)

	kanonische Route		isReachable Pattern	
Distanz	# Transitknoten	Zeit [h]	# Transitknoten	Zeit [h]
2	916.267	0,16	911.627	0,25
3	793.207	11,28	782.138	20,31
4	718.716	132	701.447	250

Tabelle A.22.: Transitknotenbestimmung über vorherige VC Auswahl (FR)

	DE Graph		FR Graph	
Distanz	Schranke	Zeit [h]	Schranke	Zeit [h]
2	488.810	0,15	436.596	0,15
3	303.400	9,29	270.437	10,86
4	211.285	130	187.846	160

Tabelle A.23.: Untere Schranke für disjunkte kürzeste Wege

B. Dokumentation

Das Programm besteht aus einem jar Archiv, das mit dem Befehl

java -Xms8g -Xmx8g -Xss1g -jar wikipediaLinkgraph.jar [*argument*]

gestartet wird. Die einzelnen Parameter haben die folgenden Funktionen:

- Xms: Spezifiziert die Anfangsgröße des Java Heaps und sollte idealerweise genauso groß gewählt werden wie Xmx, da somit kostspielige Vergrößerungen der Java Heap zur Laufzeit vermieden werden.
- Xmx: Spezifiziert die Maximalgröße des Java Heaps und benötigt für die englische Wikipedia eine Mindestgröße in Höhe von 6GB. Damit jedoch der garbage collector nicht zu häufig arbeiten muss, empfiehlt sich hier eine Größe von 8 GB.
- Xss: Beschreibt die Größe des Thread Stacks, welcher auf 1GB gesetzt werden sollte, da einige Rekursionsaufrufe die Standardgröße des Stacks übersteigen können.
- [argument]: Optionales Argument um eine andere Konfigurationsdatei als den Standard zu definieren.

Um eine größtmögliche Plattformunabhängigkeit zu erreichen, wurde auf plattformabhängige Implementierungen und Bibliotheken verzichtet. Dadurch kann es jedoch vorkommen, dass bei langen Statusmeldungen für jeden neuen Status eine neue Zeile geschrieben wird (anstatt die aktuelle Zeile zu aktualisieren). Dies kann durch ein breiteres Fenster verhindert werden.

B.1. Konfiguration

Bei jedem Programmaufruf, bei der die Standardkonfigurationsdatei verwendet wird, jedoch keine vorliegt, wird automatisch eine Konfigurationsdatei in den selben Pfad generiert. Diese Datei definiert alle wichtigen Programmvariablen und Pfade, sodass die Korrektheit der Werte in dieser Datei sicher gestellt werden müssen. Diese automatisch generierte Datei erstellt alle Pfadangaben relativ zum aktuellen Pfad. Tabelle 2.4).
Falls unterschiedliche Wikipedia Versionen untersucht werden sollen, besteht die Möglichkeit, eine andere Konfigurationsdatei als [argument] beim Programmstart zu übergeben.

Einige Erläuterungen zu den verschiedenen Schlüsseln:

- path_x : Pfadangabe hinführend zu x.
- file_range: Wichtig für das Serialisieren der kürzesten Wege, legt die Anzahl der Startknoten pro Datei fest.
- thread_interval: Intervallangabe für Threads. Jeder Thread arbeitet diese Anzahl an Knoten ab, bevor ein neuer Arbeitsschritt begonnen wird.

- console_output: Häufigkeit der Aktualisierungen der Statusanzeige.
- threads: Anzahl der Threads, die genutzt werden sollen.
- default_nodeId: Die standardmäßig definierte Ziel Id für eine Suche.
- default_edge_direction: Die standardmäßige Richtung der Kanten (*out* oder *in*).
- path_fastLoad: Pfad für das binäre Einlesen und Speichern.
- vc_default_algorithm: Der standardmäßige Vertex Cover Algorithmus (*Greedy* oder *2-APX*), der genutzt werden soll.

B.2. Menüpunkte

Das Menü besteht aus einem übergeordneten Menü, in der die Auswahl der aufgezählten Menüpunkte über die eingegebene Zahl erfolgt. Menüpunkt [100] öffnet ein Untermenü, in der zusätzliche Punkte (wie beispielsweise Id/Titel Konvertierung oder das Extrahieren eines bestimmten Zeilenbereichs einer Datei) ausgewählt werden können. Die Menüstruktur sollte selbsterklärend sein. Dennoch seien für die Menüpunkte folgende Hinweise und Erläuterungen gegeben:

- [0]: Beendet die Anwendung.
- [1]: Manuelle Garbage Collection mit Ausgabe der Speicherinformationen.
- [2]: Extrahiert die Knoten Ids aus dem Namensraum 0, siehe Abschnitt 2.1.
- [3]: Sortiert die extrahierten Knoten Ids, siehe Abschnitt 2.1.
- [4]: Schreibt die Verweise als Strings in eine temporäre Datei, siehe Abschnitt 2.1.
- [5]: Parst die Verweise aus [4] an die Knoten Ids aus [3], wodurch die ausgehende Kantenliste erstellt wird.
- [6]: Extrahiert die eingehende Kantenliste in eine Datei, siehe Abschnitt 2.3.
- [2-6]: Sollten in dieser Reihenfolge abgearbeitet werden. Das Programm kann dazwischen unterbrochen werden, jedoch sollte das Ergebnis des vorherigen Abschnittes vorliegen.
- [9]: Deallokation nahezu aller allokierten Bereiche.
- [10]: Lädt den Graph ohne Titel-zu-Id und ohne Id-zu-Titel Zuordnung in den Speicher.
- [11]: Lädt den Graph ohne Titel-zu-Id und mit Id-zu-Titel Zuordnung in den Speicher.
- [12]: Lädt den Graph mit Titel-zu-Id und ohne Id-zu-Titel Zuordnung in den Speicher.
- [13]: Lädt den Graph mit Titel-zu-Id und mit Id-zu-Titel Zuordnung in den Speicher.
- [14]: Berechnet Knoten mit der höchsten Anzahl ein- und ausgehender Kanten, siehe Tabelle 3.4.
- [15]: Berechnet die kürzeste Weglängen, siehe Abbildung 3.3.
- [16]: Berechnet Anzahl der Knoten mit angegebener Distanz für alle Knoten.
- [17]: Berechnet Anzahl der Knoten mit angegebener Distanz für einen Knoten.
- [18]: Berechnet Anzahl der kontrahierbaren Kanten, siehe Abschnitt 3.4.

- [19]: Berechnet Knotenvalenz, siehe Abschnitt 3.1.
- [20]: Liefert die ausgehenden Kanten für den angegebenen Knoten.
- [21]: Liefert die eingehenden Kanten für den angegebenen Knoten.
- [22]: Gerichtete Breitensuche von einem Start- zum Zielknoten.
- [23]: Gerichtete Breitensuche von einem Start- zum Standardzielknoten.
- [24]: Gerichtete Breitensuche vom Startknoten aus, stoppt nicht bis der ganze erreichbare Graph traversiert wurde.
- [25]: Gerichteter Breitensuche Benchmark, siehe Abschnitt 4.2.
- [30]: Liefert die ungerichteten Kanten für den angegebenen Knoten.
- [32]: Ungerichtete Breitensuche von einem Start- zum Zielknoten.
- [33]: Ungerichtete Breitensuche von einem Start- zum Standardzielknoten.
- [34]: Ungerichtete Breitensuche vom Startknoten aus, stoppt nicht bis der ganze erreichbare Graph traversiert wurde.
- [40]: Berechnet die schwachen Zusammenhangskomponenten, siehe Abschnitt 3.2.
- [41]: Berechnet die starken Zusammenhangskomponenten, siehe Abschnitt 3.2.
- [50]: Schreibt die kürzesten Wege (ohne Alternativwege und auf Basis ausgehender Kanten) für die angegebene Distanz auf, siehe Abschnitt 5.1.
- [51]: Schreibt die kürzesten Wege (ohne Alternativwege und auf Basis eingehender Kanten) für die angegebene Distanz auf, siehe Abschnitt 5.1.
- [52]: Multithreading Variante von [50], siehe Abschnitt 5.1.
- [53]: Multithreading Variante von [51], siehe Abschnitt 5.1.
- [60]: Vertex Cover mit 2-APX berechnen, siehe Unterabschnitt 6.4.1.
- [61]: Vertex Cover mit Greedy berechnen, siehe Unterabschnitt 6.4.2.
- [62]: Überprüft, ob das Vertex Cover gültig ist.
- [63]: Berechnet Transitknotenset, bei der Knoten für die Untersuchung zufällig ausgewählt werden. Nutzt die kanonische Route Abschnitt 6.2.
- [64]: Berechnet Transitknotenset, bei der Knoten für die Untersuchung zufällig ausgewählt werden. Nutzt das isReachable pattern Abschnitt 6.2.
- [65]: Berechnet Transitknotenset, bei der Knoten mit hohem Grad zuerst untersucht werden. Nutzt die kanonische Route, siehe Abschnitt 6.3.
- [66]: Berechnet Transitknotenset, bei der Knoten mit hohem Grad zuerst untersucht werden. Nutzt das isReachable pattern, siehe Abschnitt 6.3.
- [67]: Berechnet Transitknotenset mit Hilfe des VC und der kanonischen Route, siehe Abschnitt 6.4.
- [68]: Berechnet Transitknotenset mit Hilfe des VC und des isReachable patterns, siehe Abschnitt 6.4.

- [69]: Überprüft, ob das Transitknotenset gültig ist.
- [70]: Sucht eine untere Schranke für das Transitknotenset, siehe Abschnitt 6.5.
- [91]: Serialisiert den im Speicher befindlichen Graph. Wichtig für schnelles Einlesen, siehe Tabelle 2.5.
- [92]: Liest den serialisierten Graph ein. Die meisten Menüpunkte versuchen dies automatisch.
- [93]: Serialisiert die Transitknoten in eine Datei.
- [94]: Lädt die Transitknoten aus einer Datei.
- [99]: Aktualisiert die Konfiguration.
- [100]: Öffnet/Schließt das Untermenü.
- [101]: Liefert die Id für einen Titelstring.
- [102]: Liefert den Titelstring für eine Id.
- [103]: Extrahiert angegebenen Bereich einer Datei.

Für die Berechnung der Transitknotenmenge ([63-68]) nutzen einige Algorithmen, die vom isReachable pattern Gebrauch machen, mehrere Threads. Dies hat zur Folge, dass das Ergebnis nicht optimal sein kein, da das IsReachable pattern bei jedem setzen eines Transitknotens für jeden einzelnen Thread geupdated werden müsste. Aus Effektivitätsgründen wird dies jedoch nicht gemacht. Es kann jedoch durchaus sein, dass die Threaded Variante trotzdem bessere Ergebnisse liefert, da die Knoten in einer anderen Reihenfolge abgearbeitet werden. Falls trotzdem nur ein Thread genutzt werden soll, muss der Wert des Schlüssels *threads* in der Konfigurationsdatei entsprechend auf 1 gesetzt werden.

B.3. Der erste Start

Beim erstmaligen Ausführen erstellt das Programm im selben Pfad in der sich auch die Anwendung selbst befindet, eine Konfigurationsdatei. Es muss sichergestellt werden, dass alle Pfade in dieser Datei korrekt sind. Im Allgemeinen werden die beiden Wikipedia Quelldateien *page* und *pagelinks* (siehe Tabelle 2.3) in einem Unterordner *wiki* als *page.sql* und *pagelinks.sql* erwartet. Darüber hinaus werden alle weiteren standardmäßig definierten Pfade entweder im Programmausführungspfad oder in dem Unterordner *wiki* gesetzt.
Nach diesem Schritt sollte die Wikipedia-Linkgraph Extraktion ausgeführt werden, welche durch die Menüpunkte [2] bis [6] (in dieser Reihenfolge) realisiert wird. Nach diesem Schritt steht die volle Funktionalität der Anwendung zur Verfügung.
Für eine schnellere Einlesephase sollte nach einem initialen Einlesen der Graphstruktur durch die Menüpunkte [10],[11],[12] oder [13] der Graph in binärer Form einmalig durch [91] gespeichert werden (siehe Abschnitt 2.4).
Nun kann die Anwendung anhand der Menüstruktur (siehe auch Abschnitt B.2) verwendet werden.

Literaturverzeichnis

[AHU83] A. V. Aho, J. E. Hopcroft, J. D. Ullman. *Data structures and algorithms*. Addison Wesley, 1983.

[BD09] R. Bauer, D. Delling. SHARC: Fast and Robust Unidirectional Routing. Invited submission to a special issue of the ACM Journal of Experimental Algorithmics devoted to the best papers of ALENEX 2008, 2009.

[BFM+07] H. Bast, S. Funke, D. Matijevic, P. Sanders, D. Schultes. In Transit to Constant Time Shortest-Path Queries in Road Networks. In *ALENEX*. 2007.

[BFM09] H. Bast, S. Funke, D. Matijevic. Ultrafast shortest-path queries via transit nodes. In C. Demetrescu, A. V. Goldberg, D. S. Johnson, Herausgeber, *The shortest path problem : ninth DIMACS implementation challenge*, Band 74 von *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, S. 175–192. AMS, Providence, RI, 2009.

[DGNW11] D. Delling, A. V. Goldberg, A. Nowatzyk, R. F. Werneck. PHAST: Hardware-Accelerated Shortest Path Trees. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, S. 921–931. IEEE Computer Society, Washington, DC, USA, 2011.

[Dij59] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[DW07] D. Delling, D. Wagner. Landmark-Based Routing in Dynamic Graphs. In *Experimental Algorithms*, Band 4525 von *Lecture Notes in Computer Science*, S. 52–65. Springer Berlin Heidelberg, 2007.

[EF12] J. Eisner, S. Funke. Transit Nodes - Lower Bounds and Refined Construction. In *14th Meeting on Algorithm Engineering and Experiments (ALENEX)*, S. 141–149. 2012.

[FT84] M. Fredman, R. Tarjan. Fibonacci Heaps And Their Uses In Improved Network Optimization Algorithms. In *Foundations of Computer Science, 1984. 25th Annual Symposium on*, S. 338 – 346. 1984.

[GH05] A. V. Goldberg, C. Harrelson. Computing the shortest path: A search meets graph theory. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '05, S. 156–165. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2005.

[GJ79] M. R. Garey, D. S. Johnson. *Computers and intractability*. W. H. Freeman and Co., San Francisco, Calif., 1979. A guide to the theory of NP-completeness, A Series of Books in the Mathematical Sciences.

[GSSD08] R. Geisberger, P. Sanders, D. Schultes, D. Delling. Contraction hierarchies: faster and simpler hierarchical routing in road networks. In *Proceedings of the 7th international conference on Experimental algorithms*, WEA'08, S. 319–333. Springer-Verlag, Berlin, Heidelberg, 2008.

[Hasoo] S. Hasselberg. *Some results on heuristical algorithms for shortest path problems in large road networks*. Dissertation, University of Cologne, 2000. URL `http://nbn-resolving.de/urn:nbn:de:hbz:38-6237`.

[HNR68] P. Hart, N. Nilsson, B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100 –107, 1968.

[KMS05] E. Köhler, R. H. Möhring, H. Schilling. Acceleration of shortest path and constrained shortest path computation. In *Proceedings of the 4th international conference on Experimental and Efficient Algorithms*, WEA'05, S. 126–138. Springer-Verlag, Berlin, Heidelberg, 2005.

[Lau97] U. Lauther. Slow Preprocessing of Graphs for Extremely Fast Shortest Path Calculations. Lecture at the Workshop on Computational Integer Programming at ZIB, 1997.

[Lau04] U. Lauther. An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. In *Geoinformation und Mobilität – von der Forschung zur praktischen Anwendung*, Band 22, S. 219–230. 2004.

[Moo59] E. F. Moore. The shortest path through a maze. In *Proc. of the International Symposium on the Theory of Switching*, S. 285–292. Harvard University Press, 1959.

[MSS+05] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, T. Willhalm. Partitioning Graphs to Speed Up Dijkstra's Algorithm. In *WEA*, Lecture Notes in Computer Science, S. 189–202. 2005.

[MSS+06] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, T. Willhalm. Partitioning graphs to speedup Dijkstra's algorithm. *J. Exp. Algorithmics*, 11, 2006.

[Ros08] J. Rose. fixnums in the VM, 2008. URL `https://blogs.oracle.com/jrose/entry/fixnums_in_the_vm`. [Online; Stand 01. März 2013].

[Sha81] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics With Applications*, 7:67–72, 1981.

[Wik13a] Wikipedia. Data dumps — Wikipedia, The Free Encyclopedia, 2013. URL `http://meta.wikimedia.org/w/index.php?title=Data_dumps&oldid=4308724`. [Online; Stand 01. März 2013].

[Wik13b] Wikipedia. "Wikipedia — Wikipedia, The Free Encyclopedia, 2013. URL `en.wikipedia.org/w/index.php?title=Wikipedia&oldid=539847537`. [Online; Stand 01. März 2013].

[Wik13c] Wikipedia. Wikipedia:Database download — Wikipedia, The Free Encyclopedia, 2013. URL `http://en.wikipedia.org/w/index.php?title=Wikipedia:Database_download&oldid=523844499`. [Online; Stand 01. März 2013].

Alle URLs wurden zuletzt am 01. 03. 2013 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben.
Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet.
Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens.
Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht.
Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

(Ferdi Kara)