Bachelorarbeit

im Studiengang Computerlinguistik

an der Ludwig- Maximilians- Universität München

Fakultät für Sprach- und Literaturwissenschaften

Entwicklung eines Rankingverfahrens der Suchtreffer für die FinderApp WiTTFind im Nachlass Ludwig Wittgensteins

vorgelegt von
Florian Babl

Betreuer:	Dr. Maximilian Hadersbeck
Prüfer:	Dr. Maximilian Hadersbeck
Bearbeitungszeitraum:	1. April - 11. Juni 2019

Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig angefertigt, alle Zitate als solche kenntlich gemacht sowie alle benutzten Quellen und Hilfsmittel angegeben habe.

München, den 11. Juni 2019

.....
Florian Babl

Abstract

Ziel der vorliegenden Bachelorarbeit ist es, für die FinderApp WiTTFind ein Rankingverfahren zu entwickeln, das dazu dienen soll, Mehrwortanfragen zu verarbeiten und die Ergebnisse nach ihrer Relevanz zu sortieren. Dabei wurde die schon existierende Suche der WiTTFind-Suche so optimiert, dass zu jedem Satz und damit auch Absatz, ein Score berechnet wird. Der Score dient als eine Aussage darüber, wie relevant der Suchtreffer für den Suchenden ist. Zusätzlich wurde im Rahmen dieser Arbeit den Nutzern der Suche ermöglicht relevante Ergebnisse zu einer Suchanfrage zu liefern. Die Ergebnisse müssen dabei nicht wortwörtlich und hintereinander im Wittgenstein Nachlass vorkommen. Für das Rankingverfahren wird eine eigens erstellte Formel für die Berechnung der Scores vorgestellt. Sie beinhaltet mehrere selbst gewählte und implementiere Features, wie Distanz zwischen Matches, Lemmata und Matchgenauigkeit. Ein wesentlicher Bestandteil des Rankings sind lineare Belohnungen und Bestrafungen. Am Ende der Arbeit werden die Ergebnisse vorgestellt, die durch die neue Suche erreicht werden. Teil der Arbeit ist es auch, einen unkomplizierten Einstieg die Programmierung des *wf* zu ermöglichen. Zusätzlich wird kurz die Entstehung und Funktionalität der neuen Wittgenstein Advanced Search Tools Dokumentation erklärt. Die Arbeit ist für jeden interessant, der sich einen Einblick in die Erstellung eines Rankingverfahrens verschaffen möchte und ein besseres Verständnis der WiTTFind Suche erstrebt.

Inhaltsverzeichnis

1 Einleitung

Als Nutzer einer Websuche ist man es durch Google gewöhnt, einfach mehrere Begriffe hintereinander zu nennen und als Folge relevante Suchergebnisse geliefert werden. Das Ergebnis ist dabei immer stichhaltig und im Bezug zur Frage. Dadurch verändert sich in der Konsequenz die Einstellung in Bezug auf andere Suchmaschinen. Die WiTTFind Suchmaschine durchsucht einen Korpus und soll nicht die Funktion einer Suchanfrage wie Google erkennen können. Das ist auch gar nicht ihr Zweck. Denn sie soll Textstellen aus dem Nachlass von Ludwig Wittgenstein finden, die relevant für die Suchanfrage sind. Bei einer Suchanfrage für zwei oder mehr Begriffe, möchte der Nutzer Textstellen, in denen möglichst alle Begriffe vorkommen. Jedoch ist die Suche momentan limitiert darauf Begriffe zu finden, die direkt aufeinanderfolgen. Das Problem lässt sich am besten anhand eines Beispiels erklären. In der WiTTFind-Suchanfrage "Gedanken Welt" werden keine Ergebnisse zurückgeliefert, die für den User möglichst relevant sein können. Da die Wortkombination "Gedanken Welt" im Korpus an keiner Stelle direkt hintereinander auftritt, würde die aktuelle Suchmethode keine relevanten Ergebnisse ausgeben. Dennoch gibt es Sätze in Wittgensteins Nachlass, in denen "Gedanken" und "Welt" in einem Satz vorkommen, jedoch nicht unmittelbar. Die Problemstellung ist es dementsprechend eine Methode zu entwickeln, mit der diese direkte Reihenfolge nicht mehr von Nöten ist, um Treffer zu finden. Nichtsdestotrotz soll dabei gewährleistet sein, dass Begriffe, die aufeinander folgen als Erstes in den Ergebnissen auftauchen. Für so ein Ranking muss also jeder Treffer mit einem Score ausgestattet werden, der beschreibt, wie relevant ein Treffer ist.
Ein Patent aus dem Jahr 2003 von Google Developer Krishna Bharat liefert eine Möglichkeit, einen solchen Score zu berechnen. Die Umstände und Ziele von Google sind nicht mit dieser Arbeit zu vergleichen, jedoch wird klar, dass auch im Patent von einem Ranking nach Relevanz die Rede ist. Es wird außerdem jedem Hit, d. h. jeder Website, ein Rank zugeordnet und nach dem höchsten Rank sortiert [4][5 ff.]. Für diese Arbeit soll der Fokus darauf liegen, eine Formel mit Features zu entwickeln, die für die Zwecke von WiTTFind relevant sind. Die Ranks sollen dann den Hits zugewiesen werden bevor sie anschließend sortiert werden.

2 Erstellung der WAST Dokumentation mit Sphinx

2.1 Motivation

Das Wittgenstein Advanced Search Tools (WAST), Projekt läuft seit mehreren Jahren unter der Leitung von Dr. Max Hadersbeck und Alois Pichler. Obwohl das Projekt in der Vergangenheit bereits im Rahmen diverser Bachelorarbeiten um einige Features erweitert und verfeinert wurde, konnte die vollständige Dokumentation des Projekts bisher nocht nicht abgeschlossen werden. Wie den letzten Aufzeichnungen von Dr. Maximilian Hadersbeck und Daniel Bruder zu entnehmen ist, war die letzte Aktualisierung auf den neuesten Stand im Januar 2014 [3, 1]. Eine umfassende Dokumentation ist jedoch essenziell für eine schnelle und erfolgreiche Einarbeitung in das gesamte bzw. eines der mittlerweile zahlreichen Teilprojekte.
Für jede der angesprochenen Bachelorarbeiten und für jeden Bestandteil von WAST wird im entsprechenden Gitlab Ordner ein neuer Branch bzw. ein neues Projekt erstellt. Hierbei ist der Ablauf fest vorgeschireben und bewirkt, dass für jedes Teilprojekt eine README.md Datei erstellt wird. Der Aufbau von Gitlab sorgt jedoch dafür, dass die Suche nach speziellen Begriffen keine Ergebnisse liefert. Deshalb ist die Erstellung einer eigenen Dokumentation, die alle Projekte umspannt, vonnöten. Von zentraler Bedeutung ist dabei, dass die entstehende Übersicht online auf ansprechende Art und Weise eingebunden werden kann. Darüber hinaus soll die Dokumentation auch einfach zu durchsuchen sein. Wie ein Vorlesungsskript sollte man nach bestimmten Begriffen suchen können und alle Ergebnisse angezeigt bekommen. Die Bachelor und Master Studenten finden dadurch Zeit sparender und effektiver einen Zugang zum WAST Projekt.

2.2 Sphinx Tool

Zur Erstellung der Dokumentation wurde das Sphinx Tool ausgewählt. Sphinx ist ein Werkzeug zur Softwaredokumentation, das erstmals 2008 aufgetreten ist. Sphinx nutzt zum Generieren einer Website oder einer PDF Datei reStructuredText oder Markdown Dateien. Dabei ist Sphinx vor allem für die gesamte Dokumentation des Python-Projekts bekannt geworden, welche komplett mit Sphinx realisiert wurde [11]. Ferner wurde das Programm zur Dokumentation anderer Python Module genutzt und sogar zum Schreiben von Büchern genutzt [9]. Sphinx erweist sich dank seiner weitreichenden Möglichkeiten als noch hilfreicher als Latex, um eine Dokumentation zu erstellen und ermöglicht es, die bestehenden Teil-Dokumentationen des WAST Projekts zusammenzufassen. Alle in 2.1 erwähnten Ziele können folglich mit Sphinx erreicht werden.

2.3 Funktionalität

Mithilfe von Sphinx und der Hilfe der READMEs aus wf, sis3, witt-data uvm. kann eine Website erstellt werden, die es möglich macht, alle READMEs auf einmal zu durchsuchen und damit die wichtigsten Treffer anzuzeigen. Die einzelnen README Dateien können im Verzeichnis “WAST-DOC/source” gefunden werden. An den nachfolgenden Ablauf ist sich zu halten, sobald der Dokumentation neue Markdown Dateien hinzugefügt oder die Ansicht der Website abgeändert werden soll:

Zuerst muss wird der Branch "CAST/infrastructure/wast/tree/WAST_Dokumentation/WAST-DOKU" aus Gitlab geclont, Sphinx, recommonmark und Sphinx-Markdown-Tables mit "pip" oder "pip3" installiert. Danach wird in den entsprechenden Ordner gewechselt und `make html` aufgerufen. Nachdem nun alle Markdown Dokumente verarbeitet wurden, kann jetzt die File "index.html" aud dem "build" Ordner mit Firefox geöffnet werden (Seite 52). Die Datei config.py musste um das Modul *CommonMarkParser* und *sphinx_markdown_tables* erweitert werden, damit Markdown Dateien auch als solche geparst werden können und Tabellen erkannt werden. Für eine verbesserte Ansicht der Website sind zusätzlich die Überschriften aus den Markdown Dateien angepasst worden, denn der CommonMarktParser macht aus allen Level eins Überschriften einen Unterpunkt.

```
1  from recommonmark.parser import CommonMarkParser
2
3  source_parsers = {
4      '.md': CommonMarkParser,
5  }
6
7  source_suffix = ['.rst', '.md']
8  ...
9  ...
10 extensions = [
11     'sphinx_markdown_tables',
12 ]
13 ...
```

2.4 Ergebnis

Zusätzlich zu den bereits genannten Features, bietet Sphinx außerdem die Möglichkeit, das Design der Website frei zu wählen. Die fertige Internetseite ist im nachfolgenden Bild zu sehen. Neben der Möglichkeit, alle links aufgelisteten Teilprojekte zu durchsuchen, existieren auch noch Unterpunkte für die einzelnen Stichpunkte innerhalb des Projekts. Für die grafische Oberfläche sorgt "sphinx_rtd_theme". Ein anderes Design-Template kann in der Datei *config.py* eingestellt werden.

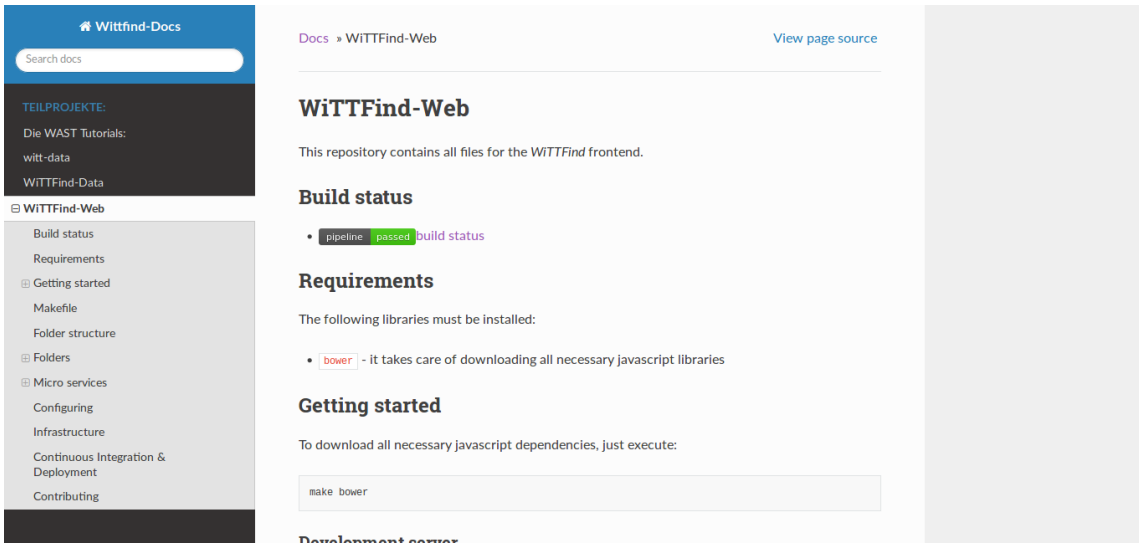

Abbildung 2.1: Ansicht der neuen WAST Dokumentationswebsite

In den nächsten Schritten muss die Dokumentation nun online angeboten werden. Des Weiteren wäre es sinnvoll, ein Skript zu schreiben, das einmal im Monat ausgewählte README Dateien aus dem WAST-Gitlab liest, aktualisiert und dadurch die Dokumentation stets aktuell hält.

3 wf: Aktueller Stand

Derzeit stehen bereits viele Suchmöglichkeiten in wf zur Verfügung. So kann zum Beispiel nach einem oder mehreren Begriffen gesucht werden. Die Suche ermöglicht es dem Nutzer, mit regulären Ausdrücken zu arbeiten oder auch nach Wortart-Tags zu suchen.Auch existiert eine Methode, die Partikelverben finden kann. Des Weiteren gibt es unterschiedliche Arten der Suche. Beispielsweise wird genau der String α gefunden, wenn dieser in Hochkommata gesetzt wird. Der Code ist zwar außerordentlich strukturiert und leserlich, jedoch fehlt es an Kommentaren oder gar einer gesamten Dokumentation. Weshalb ein guter Einstieg in den Code des Programms zu finden, erschwert ist. Im Folgenden sollen die wichtigsten Funktionen von wf erläutert werden. Dies soll einen schnellen Einstieg in das Thema ermöglichen und ein paar Begriffe und Termini erläutern, die in den anschließenden Kapiteln häufiger benutzt werden. Die genaue Erläuterung der Funktionsweise der Suche ist dabei jedoch nicht Teil dieser Arbeit. Vielmehr wird die Funktionalität von wf erklärt, damit wichtige Schnittpunkte in Zukunft schneller erkannt werden.

3.1 wf: Stuktur

3.1.1 Aufbau

In der unteren Abbildung soll der allgemeine Aufbau von wf deutlich werden. Der *wfa* Server übergibt die Suchanfrage an wf, welcher dann die Suche durchführt. Die Suche findet in *Search.hpp* statt und wird dort sehr komplex. Für die vorliegende Arbeit sind hauptsächlich die Dateien von 3.2 bis 3.7 von Interesse.

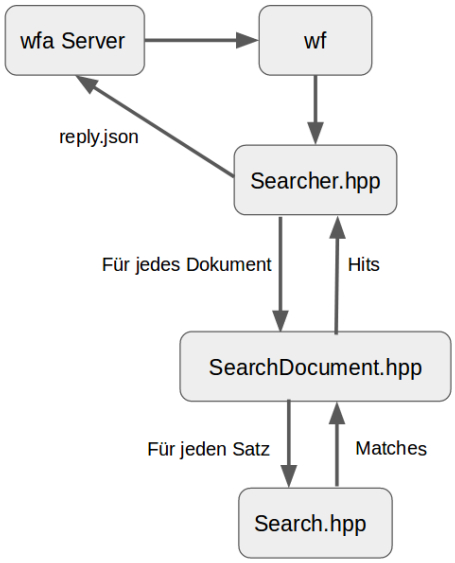

Abbildung 3.1: wf Schaubild

Wf ist eine in C++ programmierte Suchmaschine, die vom *wfa* Server aufgerufen wird und ein Json Objekt zurückliefert, das Informationen zu den gefundenen Treffern bzgl. der Suchanfrage beinhaltet (Siehe 3.2). Jedoch kann wf auch direkt von der Konsole gestartet werden. Dazu wird der *wf_ client* und *wf_ server* genutzt.

Sobald wf installiert ist, kann die Suche ausgeführt werden. Allerdings kann *wf_ client* kann nur aufgerufen werden, wenn der zuständige *wf_ server* bereits läuft. Auf Grund dessen wird dieser zuerst gestartet:

```
./bin/wf\_server --dictionary ../data/witt\_WAB\_dela\_XI.txt --
    files ../data/files.txt
```

In *files.txt* stehen die Namen der Dateien, die nach Treffern durchsucht werden. Im Anschluss wird der *wf_ client* ausgeführt:

```
./bin/wf\_client --query \"[VVFIN] \& denken\" -f ../data/Ts-213\
    _NORM-tagged.xml --max 10
```

Wenn mehrere Dateien durchsucht werden sollen, können mehrere *-f* Ausdrücke angehängt werden. Alle anderen Optionen zur Anpassung der Suche können im Anhang auf Seite 54 gefunden werden. Sobald eine neue Stelle im Code ergänzt wurde, muss im *build* Ordner der Befehl `make` aufgerufen, sowie der Server neu gestartet werden, damit die Änderungen übernommen werden.

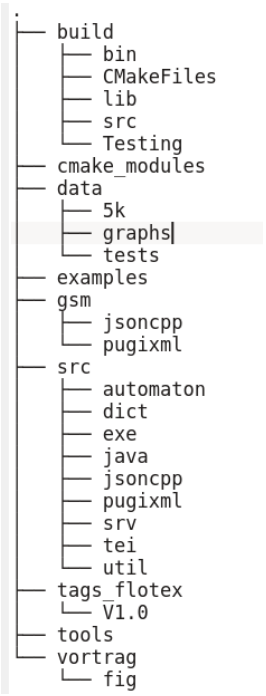

Abbildung 3.2: Verzeichnisaufbau von wf

Der Aufbau des wf Ordners ist für ein besseres Verständnis nicht zu vernachlässigen. Abbildung 3.2 zeigt die komplette Übersicht über das Verzeichnis.
Der *build*-Ordner entsteht erst, nachdem er mit `make` erstellt wurde. Aus diesem Verzeichnis heraus wird der *wf_ client* und *wf_ server* gestartet. Hier finden sich außerdem mehrere Makefiles.
Ein weiterer wichtiger Ordner ist *data*. Hier finden sich alle Text, XML, json und grf Files, die zum Testen von wf benötigt werden.
In *examples* sind sowohl die *reply.json* als auch die *request.json* zu finden.
gsm enthält die Module *pugixml* und *jsoncpp*, das zum Bearbeiten der Json Objekte unabdingbar ist.
Der gesamte Code lässt sich in den Unterverzeichnissen von *src*[1] finden. Dabei sind beson-

[1] Steht für Source.

ders sind folgende Unterordner relevant:

- *dict*: Beinhaltet den Code für den Umgang mit dem Dictionary, dem Parser und den grammatikalischen Angaben.
- *srv*: Besteht aus dem Code für die Initialisierung der tatsächlichen Suche durch den Automaten.
- *automaton*: Der Code des Automatons[2], des Handlings von Token und der Suche von Ergebnissen kann hier unter anderem recherchiert werden.
- *util*: Hier lassen sich einige utilitätsbezogene C++ Files finden.

3.1.2 Begrifflichkeiten

Um für das Verständnis ein gewisses Hintergrundwissen zu erhalten, werden in den nachfolgenden Abschnitten immer wiederkehrende Begriffe kurz näher definiert und erläutert.

- Unter Hits bzw. einem Hit wird der Treffer auf Satzebene verstanden. In Abschnitt 3.2 ist der Aufbau eines Hit-Arrays veranschaulicht.
- Für jeden Hit gibt es einen oder mehrere Matches. Jedes Match besteht aus dem Token selber und liefert einige wichtige zusätzliche Informationen für diesen Treffer.
- Ein *Query Token* ist ein Token, das in die Suchanfrage eingegeben wird. *Query-String* oder *Eingabe-String* steht stellvertretend für die gesamte Suchanfrage.
- Wenn von einem Rank die Rede ist, so ist das gleichbedeutend mit einem Score oder einem Rang. In dieser Arbeit wird mit dem Begriff "Rank" der Integerwert beschrieben, der für jeden Hit berechnet werden soll.
- In jeder der nachfolgenden und einigen hier nicht erwähnten Dateien, wurden im Code Kommentare hinzugefügt, um den Code verständlicher zu machen.
- Der neu eingeführte Key *s_rank* hält später den berechneten Rank vom Satz/Hit.
- Genauso enthält *ab_rank* den Rank vom Absatz in dem der Hit steht.
- Kurz erklärt definiert ein sog. Automaton, oder auch Automat, auf natürliche Art und Weise eine Sprache und einen zugehörigen Erkennungsalgorithmus. Der Automat bekommt ein zufälliges Wort und soll bestimmen, ob es sich um ein Wort der Sprache handelt [8][17].

3.2 Reply.json und Request.json

Request.json stellt das Json Objekt dar, das vom *wfa* Server an die wf Suche weitergeleitet wird. Reply.json repräsentiert eines der wichtigsten Objekte aus wf. Es liefert die Suchergebnisse im Json Dateiformat zurück. Eine Beispieldatei ist auf Seite 39 zu finden. Jedes Key Value Paar steht für ein wichtiges Attribut, das die Suchergebnisse genauer definiert. Die für die Bachelorarbeit relevanten Keys werden nachfolgend erläutert. Die restlichen Key Value Paare sind im Anhang auf Seite 39 zu finden.

- "buildertype": Gibt an, wie der Automat später zusammengesetzt werden soll. *SimpleBuilder* baut den Automaten so, dass zu einer Suchanfrage nur nach den Lemmata des Query Tokens gesucht wird. Hingegen findet *InverseBuilder* aber auch inverse Lemmata, d.h. wenn nach einem schon flektierten Verb gesucht wird, können jetzt das Lemma und andere Flexionen gefunden werden.

[2] Gleichbedeutend mit Automat

- "hits": Ist ein Array, das aus mehreren einzelnen Hits besteht. Jeder Hit hat folgende Keys:

  - "abnr": Gibt die Absatznummer im Dokument an.
  - "matches": Ist ein Array, das mehrere einzelne Matches enthalten kann. Jedes Match hat eigene Anmerkungen. Jedes Token hat eine spezifische ID, die in "id" übergeben wird. Der Tokenname wird in "token" gespeichert und die Position im Satz wird in "pos" abgelegt. In "gc" ist der GrammaticalCode abgelegt. Das sind grammatische Kennzeichen, die beschreiben, in welchen Flexionen und anderen Abwandlungen das Wort vorkommt.
  - "n": Jeder Satz hat ein eigenes "n". Für das Beispiel "Ts-213,iii-r[8]_1" lässt sich sagen, dass der Hit auf der Seite "Ts-213,iii-r" im ersten Satz des achten Absatzes vorkam.
  - "satznr": Für jedes Dokument sind alle Sätze durchnummeriert.

- "query": Der Value beinhaltet den Suchstring.

3.3 Searcher.hpp/.cpp

Die zwei wichtigsten Funktionen aus *Searcher.hpp* sind *search(...)* und *buildAnswer(...)*, zu finden auf Seite 37:

- In *search(...)* werden Builder, Automat und Skipper initialisiert (vgl. Zeile 3, 5,6) und für jeden Pfad zu einem Dokument die *SearchDocument()* Klasse aufgerufen (Zeile 4). Die Ergebnisse werden dann alle in der Variable *futures* gespeichert und später in das SearchDocument::Message Objekt *res* übertragen (Zeile 9 - 14). Wie man dem Code in Zeile 1 und 11 entnehmen kann, gibt es ein *request* und ein *res* [3] Objekt. Beide Objekte sind Teil verschiedener Klassen. Denn mit dem *Message* Objekt *request* wird sowohl die Suchanfrage übergeben, als auch das finale Json Objekt zurückgeliefert. Während mit dem *SearchDocument::Message* Objekt, wie in 3.5 genauer erklärt, nur die Ergebnisse abgespeichert werden. Wenn es sich nicht um eine gerankte Suche handeln soll, wird in Zeile 19 deutlich, dass im Anschluss *buildAnswer(...)* aufgerufen wird.

```
1 void wf::Searcher::search(Message& request, bool ranked_search)
    const {
2   ...
3   auto builder = request.GetBuilder(dict_); // Baut Suchautomat
      in Message.cpp
4   request.ForEachPath([&](const std::string& path) {
5     auto automaton = query->GetAutomaton(*builder);  //
        SimpleQuery.hpp gibt query token zu Automaton
6     auto skipper = request.GetSkipper(); //holt den Skipper (bei
        Satzzeichen usw.)
7     auto search = SearchDocument(        // search bekommt die
        Hits
8       std::move(automaton), std::move(skipper), files_->Get(path)
          );
9       futures.push_back(std::async(std::launch::async, std::move(
          search)));
```

[3] Steht für Results.

```
10      });
11    SearchDocument::Message res;  // vektor aus Hit Objekten
12    for (auto& future : futures) { // for each Hit
13      auto tmp = future.get(); //tmp = hits
14      std::copy(begin(tmp), end(tmp), std::back_inserter(res));
15    ...
16    if (ranked_search)
17    {
18    ...
19    }  else buildAnswer(request, res, max);
20  }
```

- *buildAnswer(...)* baut aus dem *Message* Objekt mit dem Namen *request* das endgültige Json Objekt, das am Ende von wf zurückgeliefert wird. Es werden für jeden Hit mehrere Key Value Paare hinzugefügt, die den Hit genauer beschreiben sollen, (Zeile 9). Abschließend werden die gefundenen Matches noch eingefügt (Zeile 17) und die Anzahl der Hits *n* festgelegt (Zeile: 20).

```
1  void wf::Searcher::buildAnswer(Message& request,
2                    const SearchDocument::Message& res, size_t max)
                          const {
3      ...
4      std::unordered_map<std::string, size_t> counts;
5      for (const auto& r : res) { // for each Hit
6        n += r.matches.size(); // n = Anzahl der Matches
7        for (const auto& m : r.matches) { // alle Matches iterieren
8          ...
9          auto& hit = request.AppendHit()
10                       .SetPath(r.path)
11                       .SetN(r.n)
12                       .SetF(r.f)
13                       .SetAbnr(std::stoul(r.abnr))
14                       .SetSatznr(std::stoul(r.satznr))
15                       .SetCounter(i++);
16                 ...
17         for (auto i = m.first; i != m.second; ++i)
18           hit.push_back(*i);
19     ...
20     request.SetNumberOfHits(n);
21 }
```

3.4 Message.hpp/.cpp

Message.hpp beinhaltet die drei Klassen Message, Hit und Match. Jede hat mehrere Setter- / und Getter-Funktionen, die das jeweilige Objekt bearbeiten. Die bereits in 3.2 erklärten Attribute, werden von diesen Funktionen gesetzt. Beim Erstellen eines Objekts wird immer ein *Json::Value* erstellt. Daraus entwickelt sich am Ende ein großes Json Objekt der Klasse Message, das auf alle anderen Json Values verlinkt. Das Objekt bildet dann die vollständige Antwort[4] an den *wfa* Server.

[4] reply.json

3.5 SearchDocument.hpp/.cpp

In *SearchDocument()* wird über jeden Satz der File iteriert und für jeden Satz die Automaten Suche aufgerufen. Die gefundenen Treffer werden in *matches* abgespeichert. Wenn von *matches* die Rede ist, heißt das also nichts anderes als "Treffer in einem Satz". Dementsprechend wird für jeden Satz mit Treffern eine Hit-Struct erstellt. Die *struct*-Variablen werden nun von mehreren Values gefüllt. Alle Hits werden aneinander angehängt und woraus ein Vektor voller Hits entsetht, über den in *buildAnswer(...)* iteriert werden kann.

```
wf::SearchDocument::Message
wf::SearchDocument::operator()() const {
  ...
  for (const auto& sent : file_.sentences) { //iterate sentences
    matches.clear();
    search(*automaton_, //calls automaton/Search.hpp
           begin(sent.seq), // seq = Vektor aus Token
           end(sent.seq), // Iter. on begin and end of sentence.
           *skip_,
           std::back_inserter(matches)); // append "matches"
    hits.emplace_back();
    hits.back().path = path;
    hits.back().n = sent.n; //siglum
    hits.back().f = sent.f; //facsimile
    hits.back().abnr = sent.abnr;
    hits.back().satznr = sent.satznr;
    hits.back().matches = matches;
    n += matches.size();
  } ...
}
```

3.6 Dictionary.hpp/.cpp

Jeder Eintrag im Dictionary ist zu einem *struct* Entry verlinkt. Jeder Entry kann mit der Funktion *find(...)* aus der Klasse Dictionary gefunden werden. *find(...)* kann entweder mit der id des Entrys oder mit dem *std::wstring* aufgerufen werden. Das wird in 4.4.3 noch relevant sein. Für zwei Entries kann mit der Funktion *isLemmaOf(...)* und *isInverseLemmaOf(...)* überprüft werden, in welcher Relation die Entries zueinander stehen.

```
struct Entry {  // Jedes Wort hat einen Eintrag im Dictionary mit
    mehreren Attributen
  Entry(std::wstring s, size_t i)
    : id(i)
    , str(std::move(s))
    , gc()
    , ic()
    , lemmata()
  {}
  bool isLemmaOf(const Entry* e) const noexcept;
  bool isInverseLemmaOf(const Entry*) const noexcept;

  size_t id;
  std::wstring str;
```

```
14   GrammaticalCode gc;
15   InflectionalCode ic;
16   std::set<const Entry*> lemmata;
17 };
```

3.7 Search.hpp

An dieser Stelle wird der Code sehr detailiert und es ist nicht Aufgabe dieser Bachelorarbeit intensiv auf den Code einzugehen. Kurz zusammengefasst gibt es zwei Funktionen:
Die *search(...)* Funktion ruft die *find(...)*-Funktion (Seite 37) auf. In *find(...)* wird ein Stack aufgebaut, der abgearbeitet werden soll und die *delta(...)* Funktion übernimmt die Suche. Je nachdem, mit welchen Parametern der Automat gebaut wurde, es gibt unterschiedliche *delta(...)* Funktionen, die jetzt unterschiedlich suchen. Die neu gesetzten Kommentare im Code erklären die wichtigsten Stellen.

```
1  template<class It, class OutIt>
2  OutIt
3  wf::search(const Automaton& a, It b, It e, const Skip& skip,
       OutIt out) // out = matches,
4  {          // boolean üfr ranked search        //b = begin , e = end
5    for (b = skip.skip(b, e); b != e; b = skip.skip(b, e)) {
6      auto tmp = b;
7      // Alle b abspeichern und §anschlieend distanz messen
8      if (find(a, b, e, skip))  //wenn ein match gefunden wurde.
9        *out++ = std::make_pair(tmp, b); // tmp = iterator paar.
10     else
11       ++b; // weitergehen
12   }
13   return out;
14 }
```

3.8 wf: Problemstellung

Das Hauptproblem, das auf tieferer Ebene erkennbar wird, ist, dass die wf Suche so implementiert wurde, dass sie direkt aufeinander folgende Treffer identifizieren soll. Bei einer Sucheingabe, wie *wir denken*, wird nicht automatisch ein Hit gefunden, der einen oder mehrere Wildcards zwischen den Wörter zulässt, wie z.B. *"wir sollen denken"* oder kurz *"wir * denken"*. Nur mithilfe eines regulären Ausdrucks lässt sich die Suche entsprechend einstellen. Das führt jedoch zu anderen Problemen, die im nächsten Kapitel angesprochen werden.

4 Rankingverfahren

4.1 Motivation

An wf wurden in den letzten Jahren keine größeren Änderungen vorgenommen. Es wurden immer wieder Features ergänzt, aber grundlegend wurde die Suche nicht mehr verändert. Während wf zwar makellos funktioniert, kann man es dennoch immer weiter verbessern. Ein Feature, welches es dem Nutzer ermöglicht, Treffer auf Satzebene zu finden, auch wenn die entsprechenden Wörter nicht direkt hintereinander stehen, wäre daher eine wertvolle Ergänzung für das Projekt. Suchende könnten leichter Abschnitte finden, die für sie relevant sind und und sie müssten nicht mehr auf schwierige reguläre Audrücke zurückgreifen, um die gewünschten Ergebnisse zu erhalten. WittFind wird außerdem einsteigerfreundlich, da die Suche dann eher der Googlesuche ähnelt. Da hauptsächlich Philosophen die Suche nutzen, kann es ihnen auch helfen, schneller zu Ergebnissen zu kommen. Das entwickelte Ranking basiert auf rationalen und logischen Annahmen für relevante Features.

4.2 Zielsetzung

Das Ziel dieser Arbeit ist es, dem Nutzer zu ermöglichen, Mehrwort-Anfragen in die WiTTFind App einzugeben und dabei Ergebnisse anzuzeigen, wenn auch mehrere Worte zwischen den Suchbegriffen stehen. Bei Betrachtung des Beispiels "Welt Gedanken" liefert die WiTTFind App nur einen Treffer zurück, bei dem die beiden Query Token direkt hintereinander vorkommen. Die Resultate sollten dann nach ihrer Relevanz sortiert werden. Die aufkommende Frage, was ein Ergebnis relevant macht und welche Features man für die Berechung verwenden kann, wird in 4.4 beantwortet. Als logische Konsequenz muss jeder Hit einen Rank zugeordnet bekommen, damit die Hits nach diesem Wert sortiert werden können. Bisher ist die Reihenfolge, in denen die Treffer ausgegeben werden, durch die Reihenfolge der Dokumente bestimmt.
Hinzu kommt, dass die Ranks einem Absatz zugewiesen werden sollen. Auf lange Sicht hin soll die Website der WiTTFind App ganze Absätze anzeigen und nicht nur einzelne Hits, deshalb reicht es nicht, die Scores nur satzweise zu berechnen.

4.3 Entwicklungsprozess

4.3.1 Erste Überlegungen und Probleme

Der erste Ansatz bestand darin, die komplette Suche von wf abzuändern und so anzupassen, dass Suchergebnisse nicht hintereinanderstehen müssen, damit sie gefunden werden. Obwohl wf einwandfrei funktioniert, ist es nicht trivial den richtigen Ansatzpunkt zu finden, um so fundamentale Änderungen durchzuführen. Wenn man so eine Veränderung durchführen will, muss gewährleistet sein, dass nicht die gesamte Suche zusammenbricht. Daher wurde der Ansatz schnell wieder verworfen, da sehr viele andere Komponenten an wf hängen, die davon abhängig sind, dass wf weiterhin die Ergebnisse liefert, die es bisher geschickt hat.
Eine weitere Überlegung bestand darin, eine neue, eigenständige Suche zu schreiben, die gesondert aufgerufen wird. Die Suche sollte die Treffer einzeln matchen. Hierbei wären neue Transitions und Skipper nötig gewesen, die Teil des Automaten sind. Auch dieser Ansatz wurde verworfen, da es an der Dokumentation des Codes und an der gesamten Verständlichkeit des Automaten scheiterte.

4.3.2 Finaler Ansatz

Der finale Ansatz nimmt einen einfacheren Weg, der die anderen Komponenten von wf berücksichtigt und dennoch eigenständig aufgerufen werden kann. Mit Hilfe des logischen Operators "|" sollen die Query Token separiert werden. Mit dem "|"-Operator ist wf schon in der Lage, innerhalb von einem Satz Ergebnisse zu finden, die nicht nebeneinanderstehen müssen. Der Ausdruck $\alpha|\beta$[1] liefert aber gleichzeitig Ergebnisse, die den linken Ausdruck und/oder den rechten Ausdruck beinhalten. [10][121]. D. h. mindestens einer der beiden Ausdrücke muss wahr sein, damit der ganze Ausdruck wahr wird. Das dadurch gefundene Json Objekt beinhaltet folglich Hits mit α, mit β und Hits, die beide Ausdrücke gefunden haben. Wenn Letzteres der Fall ist, lässt sich daraus schließen, dass α und β in einem Satz zusammen aufgetreten sind. Letztlich bedeutet dies für das Ranking, dass jene Hits relevanter sind.
Die Ergebnisse werden an dieser Stelle jedoch noch nicht nach ihrer Qualität bzw. Relevanz sortiert. Die Hits sind derzeit nach ihrer Reihenfolge im Korpus sortiert. Wichtig für das Ranking ist es, eindeutig festzulegen, was einen guten Hit ausmacht. Ein Treffer, der nur α gefunden hat, ist nicht besser als ein Treffer, der α und β im gleichen Satz findet. Um die Ergebnisse sortieren zu können, müssen sie dementsprechend einen Rank zugeordnet bekommen. Demnach muss jedem Hit ein Rank übermittelt werden, der mit der Hilfe von Parametern und Features berechnet wird. Dazu wird jedes Feature zuerst implementiert und berechnet, danach werden alle Werte in einer Formel zusammengeführt.
Das Ranking kann nur für mehrere Suchbegriffe angewandt werden. Wenn nach einem Begriff gesucht wird, dann sollen weiterhin einfach nur die Token ausgegeben werden, die auf den Suchstring passen. Wie auch in 4.4.1 ersichtlich werden wird, ist es ohne eine Distanz zwischen zwei gefundenen Ergebnissen nicht möglich, die Formel komplett zu berechnen.

4.4 Die Features für das Ranking

Jede Art von Machine Learning und jeder Algorithmus beruht auf Features, die für die Berechnung des Scores unumgänglich sind. Auch für das Ranking einer Suchmaschine müssen diese Charakteristika festgelegt werden.

4.4.1 Formel zur Berechnung des Rankings

$$R_S = \frac{Q}{d} \qquad Q = \sum_{m_i=1}^{n} \frac{|m|}{|q_m|}$$

Um einen Rank für einen Hit bzw. Satz R_S zu berechnen, wird der Quotient zwischen der Qualität Q und der Distanz d aller Matches berechnet. Es wird die Anzahl eines Matches m durch die Anzahl der Vorkommen des Matches im Query-String geteilt.
Zur Veranschaulichung:
Wenn der Query-String "ich denke" ist und die Matches aus einem Hit die Token:{ich, ich, denken} beinhalten, dann bedeutet das für die Formel:

$$Q = \frac{|ich|}{|q_{\text{ich}}|} + \frac{|denken|}{|q_{\text{denken}}|} = \frac{2}{1} + \frac{1}{1} = 3$$

Jedoch gibt es ein paar Belohnungen und Bestrafungen für das Ranking, die in 4.4.4 erklärt werden.
Ein Hit wird den größtmöglichen Wert für einen Rank annehmen, wenn seine Matches nebeneinander zu finden sind, die Matches genau der Suchanfrage entsprechen und es in dem Absatz, in dem der Hit vorkommt, keine anderen Hits gibt, die seinen Rank negativ

[1] α und β repräsentieren die Suchanfragen.

beeinflussen könnten. Genaueres hierzu in 4.4.5.
Gleichzeitig bekommt ein Hit den schlechtmöglichsten Rank zugeteilt, wenn es nicht alle Token aus der Query gefunden hat, die gefundenen Token zusätzlich weit auseinanderstehen und gleichzeitig nur Abwandlungen, d. h. Vollformen vom Query-Token sind. Dies wird in Kapitel 4.4.3 detailierter präsentiert.

4.4.2 Distanz zwischen Suchtreffern

Als einfachstes, aber gleichzeitig entscheidendes Merkmal gilt die Distanz zwischen zwei Matches m_1, m_2 in einem Hit. Daraus folgt logisch, dass m_1 und m_2 höher im Gesamtranking abschneiden, je näher sie zueinander stehen. Die Distanz d nimmt den kleinstmöglichen Wert an, wenn m_1 und m_2 direkt nebeneinanderstehen. Je kleiner d, desto größer ist der Wert des Ranks. In 4.4.4 wird zu sehen sein, dass der Wert der Distanz verkleinert wird, wenn die Matches sehr gut in Bezug zur Suchanfrage waren. Somit kann d noch kleiner werden und R_S einen noch größeren Wert annehmen.

Berechnung der Distanz im Code

In Zeile 2 wird bis zum vorletzten Match im aktuellen Hit iteriert und anschließend die beiden Iteratoren konstruiert. Es wird nur bis zum vorletzten Match iteriert, damit in Zeile 4 der **foll_it** die Position des letzten Matches im String annehmen kann. Die Distanz d wird für jeden Hit berechnet und infolgedessen wird die Entfernung zwischen m_1 und m_2, ... m_{n-1} und m_n aufaddiert (vgl. Zeile 6).

```
1 if (r.matches.size()>=2){
2         for(std::vector<Match>::size_type pos = 0; pos != r.
              matches.size()-1; pos++) { // iterate over all matches

3           auto curr_it = r.matches[pos].first;
4           auto foll_it = r.matches[pos+1].first;
5           ...
6           distance += tokenDistance(curr_it, foll_it);
7           ...
```

Die Funktion **tokenDistance(...)** berechnet die Distanz zwischen zwei Iteratoren. Die Iteratoren repräsentieren zwei Matches. Nachdem beide dereferenziert wurden (6), werden aus dem Token-Match die Positionen geholt und mit einem *static_cast<double>* auf einen *double* Value gecastet. Das ist nötig, damit im weiteren Verlauf auch rationale Zahlen berechnet werden können und *size_t* nur *unsigned integer* Werte halten kann, die von *size_of* zurückgegeben werden [10][122]. Wie in 3.2 und 3.4 gezeigt, hat jedes Match einen *pos* Attribut. Anschließend wird die Position des nachfolgenden Tokens vom aktuellen abgezogen und der Wert zurückgegeben.

```
1 double
2 wf::RankedSearch::tokenDistance(std::vector<Token>::
    const_iterator &curr_it,
3                                 std::vector<Token>::const_iterator &
                                      foll_it) const
4 { // Calculate distance between the two matches
5   double distance = 0.0;
6   auto token_1 = *curr_it;
7   auto token_2 = *foll_it;
8   double pos1 = static_cast<double>(token_1.pos()); // wegen
      size_t
```

```
9   double pos2 = static_cast<double>(token_2.pos());
10  distance = pos2 -pos1;
11  return distance;
12 }
```

4.4.3 Qualität

Um die Qualität Q zu berechnen, wird eine *unordered_map<...>* **query_map** erstellt (Zeile 4). Dafür muss der Query-String gesplittet werden und jedes Wort in einen *std::wstring* abgewandelt werden (Zeile 9). Das geschieht, weil die **find(...)** Funktion nur *std::wstrings* als Argument akzeptiert, um den Dictionary Entry für den wide String zu lokalisieren. Schlussendlich wird für jeden Entry die Anzahl der Vorkommen im Query-String gezählt (Zeile 11, 15). Dabei müssen auch wieder *double*-Werte verwendet werden, damit am Ende der Rank berechnet werden kann.

```
1  std::unordered_map<const wf::dict::Entry *, double>
2  wf::RankedSearch::split(std::string query, int &query_size) const
3  {
4    std::unordered_map<const wf::dict::Entry *, double>
       query_entries;
5    std::istringstream splitter(query);
6    std::string word;
7    while(splitter >> word) {
8      query_size++;
9      std::wstring wide_str = wf::util::widen(word.c_str());
10     const wf::dict::Entry * token_entry = dict_->find(wide_str);
11     if (token_entry){
12       if (query_entries.find(token_entry) != query_entries.end())
           {
13         query_entries[token_entry] += (double)1.0;
14       } else {
15         query_entries[token_entry] = (double)1.0;
16       }
17     }
18   }
19   return query_entries;
20 }
```

Die Werte für den Nenner der Formel liegen nun vor. Für den Zähler wird ähnlich vorgegangen. Zuerst wird eine **hit_map** erstellt, die eine Kopie der **query_map** ist (Zeile 2). Jedoch werden nach jedem neuen Hit die Values in der **hit_map** auf null zurück gesetzt. Dies sorgt dafür, dass für jeden Hit der R_S neu berechnet wird und nicht die Frequenzen der alten **hit_map** miteinbezogen werden (Zeile 8).

```
1  std::unordered_map<const wf::dict::Entry *, double> hit_map;
2    hit_map.insert(query_map.begin(), query_map.end());
3
4    double ab_rank = 0.0;
5    for (auto& r : res) { // üfr jeden Hit in hits
6
7      for (auto &it : hit_map){  // set each value to 0
8        it.second = 0.0;
9      }
```

```
10    ...
```

In der gleichen *for*-Schleife wie zuvor wird nun der Dictionary Entry für das aktuelle Match *cur_match_entry* gesucht und damit **fillHitMap(...)** aufgerufen (Zeile 6). Im letzten Durchlauf der Schleife wird das gleiche auch für das letzte Match im Vektor gemacht (Zeile 11). Danach wird das Match zum aktuellen Hit hinzugefügt. In Kapitel 4.6 wird dies genauer erläutert.

```
1  if (r.matches.size()>=2){
2    for(std::vector<Match>::size_type pos = 0; pos != r.matches.
       size()-1; pos++) {
3      auto curr_it = r.matches[pos].first;
4      auto foll_it = r.matches[pos+1].first;
5      const wf::dict::Entry* cur_match_entry = (*curr_it).entry();
         // Dict-Entry von Match
6      fillHitMap(query_map, hit_map, cur_match_entry);
7      distance += tokenDistance(curr_it, foll_it);
8      hit.push_back(*curr_it); // Match ühinzufgen
9      if (pos == r.matches.size()-2){ //Im Letzten durchlauf
10       const wf::dict::Entry* foll_match_entry = (*foll_it).entry
           ();
11       fillHitMap(query_map, hit_map, foll_match_entry);
12       hit.push_back(*foll_it); // add last match
13     }
14   }
```

Anschließend wird die gesamte Qualität Q berechnet und es werden noch einige Parameter festgelegt, die für spätere Belohnungen und Bestrafungen der Scores wichtig sind (Siehe 4.4.4).

```
1  bool perfect_match = true;
2          bool all_found = true;
3          double quality = 0.0;
4          for (auto it : hit_map){
5            if ((it.second < query_map[it.first]) && (it.second >
               0.0)){
6              perfect_match = false;
7              quality += it.second;
8            }else if (it.second == 0.0){
9              perfect_match = false;
10             all_found = false;
11             quality -= query_map[it.first] * (double)2; //
                 punishment
12           }else {
13             quality += it.second;
14           }
15         }
```

Lemmata und inverse Lemmata

Der Funktion wird die **query_map**, **hit_map** und der Dictionary Entry des aktuellen Matches übergeben. Nun wird über die **query_map** iteriert. Zuerst kommt es zu einer

Überprüfung, ob es sich beim Match um ein Partikelverb handelt. Falls das der Fall ist, wird die Qualität des Hits noch einmal verbessert.

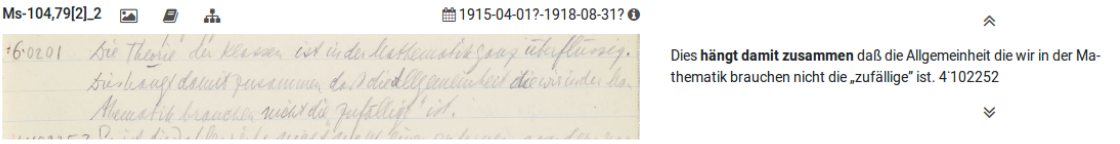

Abbildung 4.1: Suchergebnis zu Partikelverb

Wie in Abbildung 4.1 zu sehen ist, werden auch Partikelverben gefunden. Diese sollen eine Sonderrolle im Ranking einnehmen, indem sie sollen vor den anderen Ergebnissen stehen. Damit das Ranking nicht einen schlechten Wert berechnet, weil die Teile des Partikelverbs weit auseinanderstehen, wird das Ranking positiv angepasst. Das hätte zur Folge, dass das Ranking für Partikelverben sehr schlecht ausfallen würde, daher die Berichtigung (Zeile 6 ff.).
Jetzt kommt es zu einer Fallunterscheidung. Der Query Entry kann nun entweder exakt dem übergebenen Dictionary Entry entsprechen, ein Lemma, ein inverses Lemma sein oder gar nicht dem Match entsprechen. Je nachdem wird der Wert in der **hit_map** an der entsprechenden Stelle um den Wert *1.0* oder *0.9* erhöht (Zeile 10, 13, 16). Die Keys der **hit_map** bestehen aus Query Entries. Es hätte Komplikationen zur Folge, wenn man die Map mit den Dictionary Entries der gefundenen Treffer füllen würde. Die Konsequenz wäre, dass jede gefundene Vollform eines Wortes einen Eintrag in der **hit_map** bekommt. Dadurch werden keine repräsentativen Werte gebildet und die Qualität würde falsch berechnet werden. Beim Berechnen der Qualität wird von folgender Bruchregel Gebrauch gemacht: [6][73-97]
Wenn für die Anzahl an gefundenen Matches m für das Wort *denken* gilt:

$$|m| = \sum_{m_i=1}^{n} 1$$

Dann gilt:[2]

$$\frac{|m|}{|q_m|} = \sum_{m_i=1}^{n} \frac{1}{q_n}$$

```
1  void wf::RankedSearch::fillHitMap(std::unordered_map<const wf::
       dict::Entry *, double> &query_map,
2                        std::unordered_map<const wf::dict::Entry *,
                             double> &hit_map,
3                        const wf::dict::Entry * &match_entry) const {
4      for (auto query_pair : query_map){
5        auto query_entry = query_pair.first;
6        size_t vPartFound = match_entry->gc.str().find(L"+V+#");
7        if (vPartFound != std::string::npos){
8          hit_map[query_entry] += (double) 3.0;
9        }
10       if (query_entry == match_entry){
11         hit_map[query_entry] += (double)1.0 / query_map[
               query_entry];
12         break;
13       } else if ((*query_entry).isLemmaOf(match_entry)) {
```

[2] q_m ist die Anzahl an Vorkommen von m im Query-String.

```
14          hit_map[query_entry] += (double)0.9 / query_map[
                query_entry];
15          break;
16        } else if ((*match_entry).isInverseLemmaOf(query_entry)){
17          hit_map[query_entry] += (double)0.9 / query_map[
                query_entry];
18          break;
19        }
20        else { continue; }
21      }
22  }
```

4.4.4 Belohnung und Bestrafung von Parametern

Es werden des Weiteren Paramerter festgelegt, die die Formel weiter beeinflussen sollen. Zum jetztigen Zeitpunkt ist die Gewichtung der Parameter festgelegt, soll aber in Zukunft vom User festgelegt werden können (Siehe 6). In der gleichen Schleife wird dann die Qualität des gesamten Hits ausgerechnet (Zeile 7). Ein **perfect_match** ist dann gegeben, wenn der Suchtreffer genau der Suchanfrage entspricht oder optional noch mehr Token in dem Hit gefunden wurden. **All_found** soll signalisieren, dass zwar alle Wörter aus der Suche gefunden wurden, aber sie nicht exakt dem Suchtreffer entsprechen. Als Beispiel hierfür wären Vollformen oder Flexionen zu nennen, die ein Query Token teilweise erfüllen. Wenn ein Query Token nicht gefunden wurde, d.h. wenn der **it** gleich 0.0 ist, dann wird die Qualität der Suchanfrage verschlechtert (Siehe 11). Für jedes fehlendes Token wird eine Bestrafung vollzogen.

```
1   bool perfect_match = true;
2   bool all_found = true;
3   double quality = 0.0;
4   for (auto it : hit_map){
5     if ((it.second < query_map[it.first]) && (it.second > 0.0)){
6       perfect_match = false;
7       quality += it.second;
8     }else if (it.second == 0.0){
9       perfect_match = false;
10      all_found = false;
11      quality -= query_map[it.first] * (double)2; // punishment
12    }else {
13      quality += it.second;
14    }
15  }
```

r.s_rank wird mit der in 4.4.1 beschrieben Formel berechnet. Zusätzlich werden hier Belohnungen berücksichtigt, indem der Wert der Distanz kleiner gemacht wird. Außerdem bekommt der Hit seinen Rank zugewiesen (Zeile 3)

```
1 if (perfect_match){
2     r.s_rank = quality / (distance / (double)5.0);
3     hit.SetSatzRank(r.s_rank);
4   } else if (all_found) {
5     r.s_rank = quality / (distance/ (double)2.0);
6     hit.SetSatzRank(r.s_rank);
7   } else {
```

```
8       r.s_rank = quality / distance;
9       hit.SetSatzRank(r.s_rank);
10    }
```

4.4.5 Ranking über Absätze

Das Absatz-Ranking addiert für einen Absatz alle Satz-Ranks R_S zusammen und teilt den Wert durch die Anzahl der Sätze c, die mindestens einen Match hatten.

$$R_A = \frac{\sum_{i=1}^{n} R_S}{c}$$

Um ein Ranking über Absätze zu erstellen, muss eine unordered_map erstellt werden, die die Satz-Ranks pro Absatz aufsummiert und gleichzeitig mitzählt, wie oft der einzelne Absatz gefunden wurde. Demnach hat **absatz_rank** hat die Form *<std::string, std::pair<double,double»*. Um die richtige Absatznummer zu wählen, kann nicht auf das Json Attribut "abnr" zurückgegriffen werden, denn für jedes Dokument werden die Absätze erneut gezählt. Das hätte zur Folge, dass sich Dokumente und Absätze überschneiden. Daher muss auf "n" (3.2) zurückgegriffen werden. Um sowohl das Dokument als auch den Absatz aus "n" zu extrahieren, muss am Unterstrich gesplittet werden (Zeile 4). In der Funktion *getDocumentAbsatz(...)* wird genau das umgesetzt.

```
1  std::string
2  wf::RankedSearch::getDocumentAbsatz(std::string n) const{
3    std::smatch match;
4    std::regex rgx("(.*)_");
5    if (std::regex_search(n, match, rgx)){
6      return match[1];
7    } else {
8      return n;
9    }
10 }
```

Anschließend wird der Dokument-Absatz in den akutellen Hit eingetragen.

```
1  auto& hit = request.AppendHit()
2                     ...
3                     .SetDocAbsatz(docAbsatz);
```

Schlussendlich muss, nachdem der Score für jeden Satz berechnet worden ist, die *unordered_map* befüllt werden. Es wird überprüft, ob der Dokument-Absatz schon in der Map vorkommt (Zeile 6), sollte dies nicht der Fall ist, wird ein neues Paar aus **s_rank** und dem Wert 1.0 gemacht und in die Map eingfügt (Zeile 7). Im anderen Fall wird einfach **s_rank** und die Anzahl erhöht (Zeile 11 ff.).

```
1  void
2  wf::RankedSearch::fill_unord_map(std::unordered_map<std::string,
       std::pair<double, double>> & m,
3                                   std::string docAbsatz,
4                                   double s_rank) const
5  {
6    auto it = m.find( docAbsatz );
7    if ( it == m.end() ) {
```

```
8       auto value_pair = std::make_pair(s_rank,1.0); // 1.0 Anzahl
            des Absatzes
9       m.insert(std::make_pair(docAbsatz, value_pair));
10    }
11    else {
12     m[docAbsatz].first += s_rank;
13     m[docAbsatz].second += 1.0; // öErhhe anzahl der äAbstze
14    }
15  }
```

Nachdem über alle Dokumente iteriert ist, ist die *unordered_map* vollständig und kann an die Sortierung weitergegeben werden.

4.5 Sortierung

Bei der Sortierung tritt die Schwierigkeit auf, dass die klassische *sort(...)* Methode von C++ zwar ein drittes Argument zulässt mit dem die Art der Sortierung festgelegt werden kann, aber es wird kein Json::Value akzeptiert. Um noch spezifischer zu sein, *sort(...)* nimmt nur sogenannte *RandomAccessIteratoren* [1], aber beim Iterieren über ein Json::Value werden *Json::ValueIterator* verwendet. Daher muss *.json_["hits"]* in einen *std::vector* übertragen werden (Zeile 6). Dabei wird gleichzeitig der **ab_rank** für jeden Hit berechnet und gesetzt. Die Summe über alle R_S und c ist schon in 4.4.5 berechnet worden. Jetzt kann der Quotient berechnet werden und dem Key im Hit zugewiesen werden (Zeile 5).

```
1 std::vector<Json::Value> copy;
2   for (auto i : request.json_["hits"]){
3     auto key = i["DocAbsatz"].asString();
4     double ab_rank = absatz_ranks[key].first / absatz_ranks[key].
          second; //durch Anzahl der Hits von dem Absatz teilen.
5     i["ab_rank"] = ab_rank;
6     copy.push_back(i);
7   }
```

Die Kopie wird mit einem sogenannten *Compare* sortiert, mit dem festgelegt wird, nach welchem Value die Objekte angeordnet werden sollen. Es werden dabei zwei Objekte auf ihren Absatz-Rank verglichen. [2]. Der Aufruf wird von *std::sort(...)* ausgeführt.

```
1 bool
2 sortByAbsatz(const Json::Value& a, const Json::Value& b)
3 {// For sorting Hits
4     return a["ab_rank"].asDouble() > b["ab_rank"].asDouble();
5 }
```

Bevor das sortierte Array wieder an das *.json_["hits"]* Array übergeben werden kann, werden noch die einzelnen Hit Nummern "no" gesetzt und die Größe des Arrays auf die "max" Variable reduziert. Da es sich wieder um Json::Values handelt, muss der Counter mit *static_cast* auf einen Json unsigned integer gecastet werden.

```
1   Json::Value sorted_hits;
2   size_t c = 0;
3   for (auto it : copy){
4     if (c <= max) { // max wird von search.js auf 200 gesetzt
```

```
5         it["no"] = static_cast<Json::UInt>(c); // Hit-Nummer
              anpassen
6         sorted_hits.append(it); // Reduce Hits to max size
7       }
8       c++;
9     }
10    // request bekommt sortierten Vector
11    request.json_["hits"] = sorted_hits;
12  }
```

4.6 Zusammensetzung des Programms

Da bestimmte Parameter sollen später auch vom User festgelegt werden können (Siehe 6), beschreibt folgender Ablauf nur den aktuellen Stand des Programms.

Zuerst wird überprüft, ob der Query-String mindestens zwei Token enthält bzw., ob der Query-String dem Pattern in Zeile 6 entspricht. Anschließend werden die | Operatoren eingefügt und der neue Query-String gesetzt (Zeile 7).

```
1  bool
2  wf::Searcher::qualifies_for_ranked(Message& request)
3  { //Checks if user query is fitted for ranked search, if so,
       changes query to perform rankedSearch
4    std::string query = request.GetQueryStr();
5    request.oldquery = query;
6    if (std::regex_match(query, std::regex("^[\\äöüßw]+(\\s[\\äöüßw
       ]+)+$"))){
7      query = std::regex_replace (query, std::regex("\\s"), " | ");
           //changes Query to ranked searching
8      request.setRankedQueryStr(query);
9      return true;
10   } else return false;
11 }
```

Etwas weiter in der *Searcher.cpp* wird nun die gerankte Suche durchgeführt und die RankedSearch Klasse dafür initialisiert. Da *buildAnswer(...)* und *ranked(...)* jeweils die Referenz [10][97] des *request*-Message Obekts verändert, kann die Funktion einfach aufgerufen werden.

```
1  if (ranked_search)
2    {
3       RankedSearch ranked(request, res, max, dict_);
4    } else buildAnswer(request, res, max);
5  }
```

In *RankedSearch.cpp* angekommen, wird zuerst die **query_map** und **hit_map** erstellt (Siehe 4.4.3). Anschließend werden über die einzelnen Hits iteriert, die in **res** gespeichert sind (Zeile 1). Daraufhin wird für diesen Hit der Dokument-Absatz erstellt (Siehe 4.4.5). In Zeile 4 ff. wird der neue Hit mit seinen Attributen ins Array hinzugefügt.

```
1  for (auto& r : res) { // üfr jeden Hit in hits
2      ...
```

```
3       std::string docAbsatz = getDocumentAbsatz(r.n);
4       auto& hit = request.AppendHit()
5                       .SetPath(r.path)
6                       .SetN(r.n)
7                       .SetF(r.f)
8                       .SetAbnr(std::stoul(r.abnr))
9                       .SetAbsatzRank(ab_rank)
10                      .SetSatznr(std::stoul(r.satznr))
11                      .SetDocAbsatz(docAbsatz);
```

Im Anschluss wird über die Matches im aktuellen Hit iteriert, Distanz und Qualität werden berechnet, Belohnungen und Bestrafungen miteinbezogen und schlussendlich wird der Satz-Rank festgelegt, wie in 4.4.2, 4.4.3 und 4.4.4 beschrieben. Danach wird die *unordered_map* **absatz_ranks** gefüllt. Das richtige Setzen der Anzahl der Hits bildet den Abschluss des Rankingverfahrens.

```
1   request.SetNumberOfHits(request.json_["hits"].size());
```

Nachdem über alle gefundenen Hits iteriert wurde, wird die Sortierung vorgenommen, wie in 4.5 erklärt. Der sortierte Vektor wird am Ende wieder dem Hit-Array übergeben und hat damit seine finale Form.

4.7 Neues Json Objekt

Da das Ranking die Berechnung neuer Parameter erfordert, wird das Json Objekt verändern. Es werden sowohl der Rank für jeden Satz R_S, als auch der Rank für den Absatz R_A berechnet. Hinzu kommt, dass jeder Hit einen Absatznamen benötigt. Die normale Absatznummer "abnr" kann dazu nicht verwendet werden, da die Absatznummern für jedes Dokument neu gezählt werden. Daher wird das Json Objekt um drei Key Value Paare erweitert:

- "DocAbsatz" enthält den String, der aus "n" gesplittet wird. Er gibt an, in welchem Dokumentenabsatz sich der Hit befindet.
- "s_rank" enthält einen *double* Wert, da jsoncpp Probleme mit *Floats* hat. Jsoncpp schient *Floats* nicht akzeptieren zu wollen, daher die *double* Alternative. Der Key gibt Auskunft über den Satz Rank.
- "ab_rank" ist ähnlich zu "s_rank" und kann sogar den gleichen Wert annehmen. Sobald es aber mehrere Hits in einem Absatz gibt, ist das nicht mehr der Fall und ab_rank beschreibt den gesamten Absatz.

5 Ergebnisse des Rankingverfahrens

Die Ergebnisse des Rankingverfahrens sollen bestätigen, dass alle Zielsetzungen erreicht wurden. Die Suchanfragen wurden mit Hilfe von *wf_client* und *wf_server* abgefragt. Die Ergebnisse werden mit alten Suchergebnissen verglichen und sollen die Unterschiede zum neuen Ranking deutlich machen. Die alten Ergebnisse werden von der WiTTFind Website `wittfind.cis.uni-muenchen.de` bezogen. Auch wenn die Website alle Dokumente durchsucht und in den folgenden Beispielen nur fünf Dokumente berücksichtigt werden, wird durch die Bespiele dennoch deutlich, was sich verändert hat. Die einzelnen Abschnitte geben die Suchanfragen an. Zur Verdeutlichung, dass das Absatzranking auf mehreren Dateien funktioniert, wird der Client Befehl um vier weitere Dateien erweitert:

```
./bin/wf_client --query "SUCHANFRAGE" -f ../data/Ts-213\_NORM-tagged.xml -f ../data/Ts-211\_OA\_NORM-tagged.xml -f ../data/Ts-212\_OA\_NORM-tagged.xml -f ../data/Ms-142\_OA\_NORM-tagged.xml --max 20
```

Die Files müssen auch dem Ordner *data* hinzugefügt werden. Außerdem müssen die Namen in der *files.txt* Datei des Servers ergänzt werden.
Zur übersichtlicheren Darstellung des resultierenden Json Objekts wird ein *jq* Befehl genutzt:

```
| jq '.hits[] | {DocAbsatz: .DocAbsatz, ab_rank: .ab_rank, s_rank: .s_rank, matches: [{token: [.matches[].token], pos: [.matches[].pos]}]}'
```

Die Ansicht ist nur zu Zwecken der besseren und intuitiveren Verständlichkeit gewählt.

5.1 Beispiele

5.1.1 Gedanken Welt

Wie in 1 schon erwähnt, werden zu diesen Suchbegriffen keine Ergebnisse gefunden. Jetzt werden in nur fünf Dokumenten über 514 Treffer gefunden. Selbstverständlich haben nicht alle dieser Treffer einen guten Rank, aber wie man dem unteren Json Objekt entnehmen kann, besitzt der erste Hit direkt beide Treffer innerhalb eines Satzes. Die Begriffe sind auch nur drei Positionen voneinander entfernt. Daher steht der Hit an erster Stelle. Beim zweiten Treffer ist es ähnlich. Bei den Hits vier bis neun sind die Hits auf den Absatz *Ts-212,1062[1]* verteilt. Im Anhang werden auf Seite 47 die gekürzten Hits aufgelistet.

```
{
  "DocAbsatz": "Ts-212,1028[1]",
  "ab_rank": 3.3333333333333335,
  "s_rank": 3.3333333333333335,
  "matches": [
    {"token": ["Welt", "Gedanken"],
      "pos": [ 7, 10 ]
    }          ]
```

```
9  }, {
10   "DocAbsatz": "Ts-211,399[2]",
11   "ab_rank": 3.3333333333333335,
12   "s_rank": 3.3333333333333335,
13   "matches": [
14     {"token": ["Welt", "Gedanken"],
15       "pos": [ 7, 10]
16     }          ]
17 }
```

Es werden jetzt also Hits gefunden und nach ihrem Rank geordnet, die nicht unmittelbar nebeneinanderstehen und nach ihrem Rank geordnet. Die Suche wurde damit von keinem Treffer auf mehrere verbessert.

5.1.2 man weiß

Die Query Token wurden mit der alten Suche relativ oft hintereinander gefunden und sind im natürlichen Sprachgebrauch in der Reihenfolge verbreitet. Die neuen, zusätzlichen Hits sollen also nach den alten Hits angezeigt werden. Die Suchergebnisse zeigen zuerst vier perfekte Treffer, in denen beide Wörter hintereinander vorkommen. An sechster und siebter Stelle stehen wieder zwei Hits aus dem gleichen Absatz. In diesem Absatz wurden viele Wörter gefunden, die mit der Suchanfrage übereinstimmen. Daher fällt der berechnete Rank hoch aus. Hinzu kommt ein perfektes Match, bei dem beide Suchwörter wieder direkt aufeinanderfolgen. In Zeile 14 ist erkennbar, dass ein Token welches nicht dem Query Token entspricht zwar immer nohc gefunden wird, jedoch der *s_rank*-Wert niedriger ist.

```
1  {
2    "DocAbsatz": "Ts-213,55r[1]",
3    "ab_rank": 5.58,
4    "s_rank": 10,
5    "matches": [
6      {"token": ["ßwei","man"],
7        "pos": [3,4]
8      }          ]
9  }, {
10   "DocAbsatz": "Ts-213,55r[1]",
11   "ab_rank": 5.58,
12   "s_rank": 1.16,
13   "matches": [
14     {"token": ["man","wissen","man"],
15       "pos": [ 2, 4, 7]
16     }          ]
17 }
```

Das Absatzranking funktioniert also auch wie erwünscht. Zwei gute Hits innerhalb eines Absatzes bilden zusammen wieder einen guten Score für das Ranking. Ein positiver Nebeneffekt des Rankings besteht darin, dass die Reihenfolge in der die Suchbegriffe eingegeben werden keine Rolle spielt.
Bei genauerem Hinsehen finden sich dennoch Probleme. Betrachtet man die Hits auf weiter hinterliegenden Positionen, ist beispielsweise erkennbar, dass ein Hit a eigentlich einen sehr hohen *s_rank* zugewiesen bekommt. Jedoch kommt in einem anderen Satz b im gleichen Absatz nur ein Wort der Suchanfrage vor. Daraus folgt, dass das schlechte Ranking von Satz b das perfekte Ranking von Satz a zu sehr beeinflusst. Die Bestrafungen und Belohnungen müssen folglich noch genauer angepasst werden. Ferner gibt es Hits mit vielen

Matches, die aber nicht hoch genug eingestuft werden. Auf Position 97 ist folgendes Objekt zu finden:


```
{
  "DocAbsatz": "Ts-212,1812[1]",
  "ab_rank": 0.85,
  "s_rank": 0.85,
  "matches": [
    {"token": ["man","wissen","man","wisse","man","man","man"],
      "pos": [7,8,15,18,28,36,47]
    }]
}
```


Es ist der einzige Hit im Absatz *Ts-212,1812[1]*, jedoch wurde das Wort "weiß" nur in einer anderen Vollform gefunden. Der Hit wird dafür aber zu drastisch bestraft. Folglich muss noch einiges an "Finetuning" betrieben werden, um das Ranking zu optimieren. Das gesamte Ergebnis ist auf Seite 49 zu finden.

5.1.3 Haus Stadt

Bei der Suche nach "Haus Stadt" wurde bisher nur ein Ergebnis gefunden.

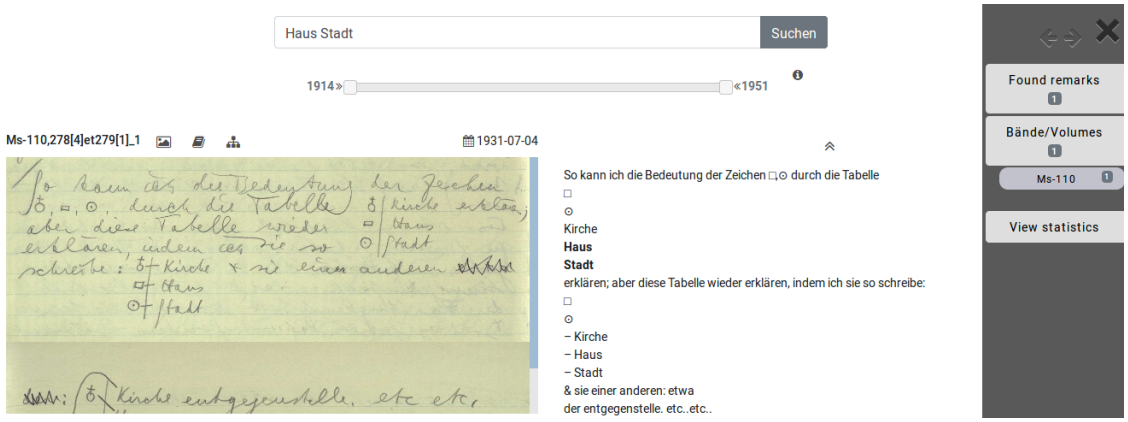

Abbildung 5.1: "Haus Stadt" Suche ohne Rankingalgorithmus

Nun werden Ergebnisse gefunden, in denen die Query Token im Kontext miteinander vorkommen. Der Rank wird schlechter, je weiter die Positionen der Matches auseinander liegen. Der erste Absatz, der in zwei Sätzen ein Match hat, ist *Ms-142,12[2]et13[1]*. Beim Betrachten der Treffer steht fest, dass der Absatz wohl zu Recht auf die ersten paar Treffer folgt.


```
{
  "DocAbsatz": "Ms-142,12[2]et13[1]",
  "ab_rank": 0.44625,
  "s_rank": 0.58,
  "matches": [
    {"token": ["äHusern","Stadt","Stadt"],
      "pos": [4,11,14]
    }]
},{
  "DocAbsatz": "Ms-142,12[2]et13[1]",
  "ab_rank": 0.44625,
```

```
12    "s_rank": 0.3125,
13    "matches": [
14      {"token": ["Stadt","äHusern","äHusern","äHusern","äHusern","
          äHusern"],
15        "pos": [8,35,50,66,69,96]
16      }]
17  }
```

Auf Seite 50 ist wieder ein größerer Teil der Suchanfrage zu finden.

5.1.4 Gott Religion Teufel

Auch längere Suchanfragen stellen kein Problem bei der Suche dar. Nur die Dauer der Suche verlängert sich, weil jeder Begriff einzeln gesucht werden muss. Die Menge der sinnvollen Ergebnisse nimmt mit zunehmender Hitnummer stark ab, weil keine Absätze mehr gefunden werden, in denen alle drei bzw. möglichst viele der Matches vorkommen. Das Beispiel soll verdeutlichen, dass auch Begriffe, die wahrscheinlich im Kontext voneinander Vorkommen, auch richtig durch die Rankingformel bewertet werden. Der untere Code-Abschnitt zeigt den am besten bewerteten Absatz. Es wurden alle drei Wörter in einem Satz gefunden und noch ein zusätzliches Wort in einem späteren Absatz.

```
1   {
2     "DocAbsatz": "Ts-211,125[7]",
3     "ab_rank": 0.5499999999999999,
4     "s_rank": 0.9333333333333332,
5     "matches": [
6       {"token": ["Religion","öGtter","Teufeln"],
7         "pos": [9,11,15]
8       }]
9   },{
10    "DocAbsatz": "Ts-211,125[7]",
11    "ab_rank": 0.5499999999999999,
12    "s_rank": 0.16666666666666666,
13    "matches": [
14      {"token": ["Teufeln"],
15        "pos": [11]
16      }]
17  }
```

Der Treffer wird dann auf folgende Art und Weise auf WiTTFind zu sehen sein.

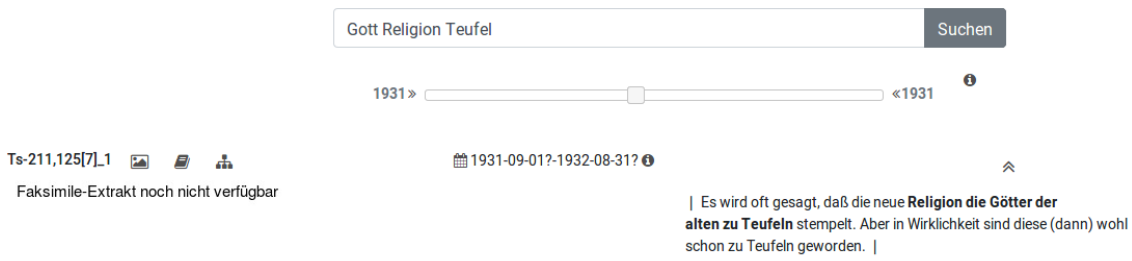

Abbildung 5.2: Ansicht des Ergebnisses zur Suchanfrage “Gott Religion Teufel”

5.1.5 Realität Vorstellung Ding selbstverständlich

Auch die Eingabe sehr langer Suchanfragen funktioniert. In diesem Beispiel soll auf die Eingliederung auf der Website aufmerksam gemacht werden. Wie die Abbildung 5.3 zeigt,

funktioniert die grafische Darstellung einwandfrei, jedoch kommt es zu Komplikationen mit dem *wfa* Server, der die Reihenfolge der Dokumente verwirft. Damit das Ranking auch die gewünschte Reihenfolge hat, muss an dieser Stelle noch etwas unternommen werden.

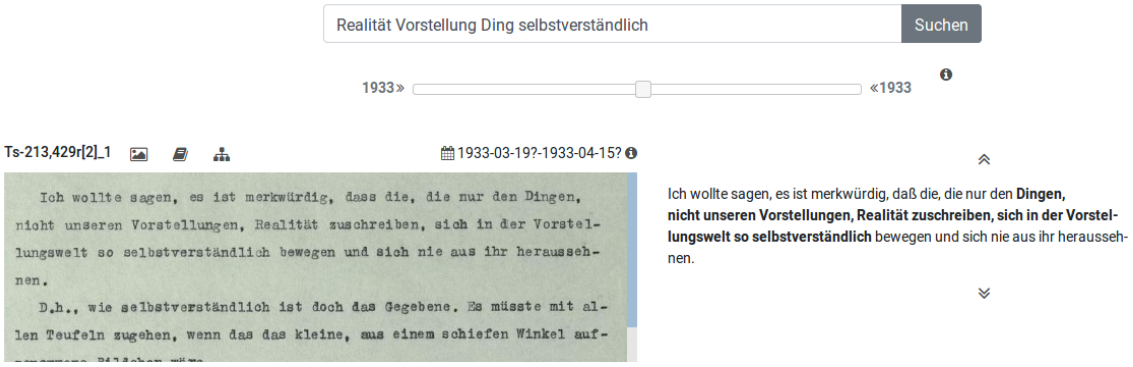

Abbildung 5.3: Ansicht des Ergebnisses zur Suchanfrage “Realität Vorstellung Ding selbstverständlich”

Der gezeigte Hit konnte hier nur durch Eingrenzen der Suche auf das Dokument *Ts-213* an die erste Position gehoben werden. Aber *wf_ client* zeigt die Ergebnisse in der richtigen Reihenfolge an. Die Sortierung funktioniert, wird aber verworfen bevor sie auf der Website angezeigt werden kann.

```
1 "DocAbsatz" : "Ts-213,429r[2]",
2 "DocAbsatz" : "Ts-213,429r[2]",
3 "DocAbsatz" : "Ts-212,1190[1]",
4 "DocAbsatz" : "Ts-212,1190[1]",
5 "DocAbsatz" : "Ts-211,763[2]et764[1]",
6 "DocAbsatz" : "Ts-211,763[2]et764[1]",
7 "DocAbsatz" : "Ts-211,763[2]et764[1]",
```

Das obere Ranking zeigt die tatsächliche Reihenfolge der Treffer.

5.2 Fazit

Das Grundgerüst für das Rankingverfahren steht und ist einsatzbereit, dennoch bietet es noch Verbesserungsmöglichkeiten. Somit wurden die formuliterten Zielsetzungen aus 4.2 erfolgreicht erreicht. Die Nutzer der Suchmaschine können jetzt Mehrwortanfragen eingeben und sich sicher sein, dass auf jeden Fall Resultate zurück kommen und, dass die Resultate nach ihrer Relevanz sortiert sind. Der Rankingalgorhithums wurde implementiert und beachtet Features wie Lemmata, Distanzen und die Qualität der Suchanfrage. Im Ergebnis funktioniert das Rankingverfahren.

6 Ausblick

6.1 Verbesserungsmöglichkeiten

In zukünftigen Arbeiten kann das Rankingverfahren und die Suche allgemein weiter verfeinert werden.

6.1.1 Einzelne Matches

Derzeit wird nur ein fester Value berechnet, wenn ein Satz nur ein Match hat, d. h. die Suchanfrage kann gar nicht zu 100% erfüllt sein. Hier kann noch eine Erkennung der Lemmata eingefügt werden. Dies führt wiederum zum Problem, dass keine Distanz ermittelt werden kann, denn dazu werden mindestens zwei Matches benötigt.

6.1.2 Frequenzlisten

Bei bestimmte Suchanfragen, wie "Gedanken über die Welt ", ist es eindeutig, dass der Suchende nach den Begriffen "Gedanken" und "Welt" sucht, als nach den beiden anderen Suchbegriffen. Derzeit wäre es jedoch so, dass alle Wörter gleich gewichtet sind. Zwar wurde mithilfe der Qualität schon der Anfang für eine Gewichtung gemacht, aber eine bessere Methode wäre es, die Informativität der Suchbegriffe zu ermitteln und diese Werte mit in die Formel einzubeziehen. Dabei kann auch von TF-IDF Verwendung gemacht werden. TF-IDF berechnet, wie relevant ein Wort in einem Dokument bzw. Korpus ist. Wörter, die nur in einer kleinen Menge an Dokumenten vorkommen, haben einen höheren TF-IDF Wert als beispielsweise Stoppwörter [7][134]. Jedoch muss so eine Datenbank zuerst aufgebaut werden und kann nicht bei jeder Suchanfrage neu berechnet werden. Wenn sie erstellt wird, kann der TF-IDF Faktor eine wichtige Rolle im Ranking einnehmen.

6.1.3 Eingliederung in die Website

Die Ergebnisse können derzeit noch nicht auf der Website angezeigt werden, da der *wfa*-Server die sortierten Ergebnisse wieder verwirft und nur bestimmte Parameter zurückliefert. Wie dem unteren Code zu entnehmen ist, übergibt *wfa* nicht das gesamte Json Objekt, das es von wf zurückbekommt.

```
1  json_response: Dict = {}
2      for item in json_wf_response["hits"]:
3          matches = item["matches"]
4          if matches:
5              date = item["date"] if "date" in item else ""
6              date_norm = item["date_norm"] if "date_norm" in item
                   else ""
7              current_triple = {
8                  "start": [matches[0]["pagenr"], matches[0]["
                       linenr"],
9                            matches[0]["tokennr"]],
10                 "end": [matches[-1]["pagenr"], matches[-1]["
                       linenr"],
11                           matches[-1]["tokennr"]],
```

```
12                      "name": item["n"],
13                      "date": date,
14                      "date_norm": date_norm}
15                  document_name = item["path"]
16                  number = item[anchored_nr]
17              ...
18      return json_response
```

Daher muss die Sortierung veraussichtlich nach *search.js* ausgelagert werden und der *wfa* Server muss die Ranks der einzelnen Hits zurückliefern. In *search.js* findet auch schon die Sortierung nach Datum statt, daher wäre es sinnvoll, auch dort die Ergebnisse nach ihrem Ranking zu ordnen. Außerdem soll dem User der Suchmaschine ermöglicht werden einzustellen, ob die Suchergebnisse gerankt ausgegeben werden sollen, oder auf die normale, alte Weise. Dazu soll auf der Website ein Knopf implementiert werden, der einen Boolean auf *True* setzt und diesen an die wf Suche weitergibt. In 4.6 ist nachzulesen, dass wf schon einen solchen Boolean besitzt, der derzeit jedoch noch anders festgelegt wird.

6.1.4 Index aufbauen für Suche

Ein Problem bei längeren Suchanfragen stellt die Dauer der Suchanfrage dar, da jeder Begriff gefunden werden muss. Um diesen Suchprozess zu verschnellern, könnte ein invertierter Index aufgebaut werden, der jedes Wort in einem Dictionary zu seinen Postings mapped [5][7]. Dadurch hätte man zu jedem Suchbegriff schon eine Liste von Dokumenten oder auch Absätzen, bevor überhaupt gesucht wird. Es müssten nicht aufs Neue alle Dokumente durchsucht werden, sondern nur noch eine limitierte Anzahl an Dokumenten. Bei mehreren Suchbegriffen könnte dann auf die Keys der einzelnen Wörter oder Teilwörter zugegriffen werden und aus den erhaltenen Dokumenten eine Vereinigung gebildet werden. Indexierung ist bei den meisten Suchmaschinen schon Standard und könnte WiTTFind auch weiter helfen, obgleich der Nachlass von Wittgenstein nicht mit der Suche im Internet vergleichbar ist.

6.1.5 Rankingstatistik

Es ist sinnvoll eine Statistik über die Verteilung der Ranks in Bezug zur Suchanfrage anzulegen. Dabei sollte man über längere Zeit Suchanfragen auf der Website tracken, um einen großen Datensatz zu haben, den man auswerten kann. Dadurch könnte man die Gewichte für die Features am besten ermitteln. Auch könnte man die durchschnittliche Länge einer Suchanfrage herausfinden und besser erkennen, wie WiTTFind genutzt wird.

6.2 Schluss

Dank Sphinx wurde eine grafisch ansprechende Website erstellt, die neuen Bachelor-/Masterstudierende einen schnelleren Einstieg in das Gesamtprojekt WAST ermöglicht. Zusätzlich ist es einfach die Website instand zu halten, da nur eine README Datei neu eingelesen werden oder neu hinzugefügt werden muss. Wie in der Arbeit aufgezeigt wird, kann man das Ranking noch weiter verfeinern, um die Suchmaschine und Ergebnisse zu verbessern, damit WiTTFind noch effektiver, schneller, handlicher und nützlicher wird. Einige der genannten Aspekte würden auch die Suche der Token so stark beschleunigen, dass man noch viel mehr Features einbauen könnte, um das Ranking weiter zu optimieren. Das Ziel, jedem Hit einen Rank zuzuordnen und nach diesem Wert zu sortieren, wurde erreicht. Außerdem kann jetzt nach Begriffen gesucht werden, die nicht zwangsweise hintereinanderstehen müssen. Die Scores für das Ranking wurden anhand eigener Features berechnet, die für das Ranking als relevant eingestuft worden sind. Es ist einfach, das Ranking nach Belieben anzupassen und

neue Gewichtungen für Features auszuprobieren. Im Ergebnis besitzt WiTTFind jetzt ein funktionierendes Mehrwort-Rankingverfahren.

7 Anhang

7.1 Code

find(...) aus Search.hpp

```
template<class It>
bool
wf::find(const Automaton& a, It& b, It e, const Skip& skip)
{
  using Pair = std::pair<It, const State*>;
  using Stack = std::stack<Pair>;
  using Set = std::set<std::pair<const State*, const Token*>>;
  Stack stack;
  Set visited;
  stack.emplace(b, a.initial()); // Fill stack  Inital() liefert
     state aus Automaton
  while (not stack.empty()) {
    auto top = stack.top(); // take first element from Stack
    auto p = std::make_pair(top.second, &(*(top.first))); //Pair
       aus state-Iterator
    stack.pop(); // Removes top element from Stack
    if (visited.count(p) == 0) { // if Element not in visited (
       set)
      visited.insert(p);  // add to visited
      if (not top.second->final()) { // if atomaton not final //
         ßHEIT?
        // top.second is autoamaton. Automaton has State.==>
           calls delta from state.cpp
        top.second->delta(top.first, e, skip, stack); // do
           something with iterator...
        //stack wird auch erweitert //
      } else { // if Word is found
        b = top.first; // b wird zu b, dass geliefert wurde.
        return true; // found something
      }
    }
  }
  return false;
}
```

7.1.1 Searcher.cpp

```
void
wf::Searcher::search(Message& request, bool ranked_search) const
{
```

```
4
5   using future = std::future<SearchDocument::Message>;
6   std::vector<future> futures;
7   auto query = request.GetQuery(); // query string
8   auto builder = request.GetBuilder(dict_); // getbuilder abstact
        query builder.// Baut Suchautomat in Message.cpp
9   assert(query);
10  assert(builder);
11
12  request.ForEachPath([&](const std::string& path) {
13    auto automaton = query->GetAutomaton(*builder); // query
          token to SimpleQuery.hpp.
14    auto skipper = request.GetSkipper(); //Skipper for
          Satzzeichen
15    auto search = SearchDocument(        // search bekommt die
          Hits
16      std::move(automaton), std::move(skipper), files_->Get(path)
          );
17
18      futures.push_back(std::async(std::launch::async, std::move(
          search)));
19  });
20  SearchDocument::Message res; // vektor aus Hit Objekten
21  for (auto& future : futures) { // for each hit
22    auto tmp = future.get(); //tmp = hits
23    std::copy(begin(tmp), end(tmp), std::back_inserter(res));
24  }
25  auto max = request.GetMax();
26  max = max ? max : std::numeric_limits<size_t>::max();
27  if (ranked_search)
28  {
29    //buildRankedAnswer(request, res, max);
30    RankedSearch ranked(request, res, max, dict_);
31  }  else buildAnswer(request, res, max);
32 }
```

```
1 void
2 wf::Searcher::buildAnswer(Message& request,
3                           const SearchDocument::Message& res,
4                           size_t max) const
5 {
6     size_t n = 0;
7     size_t i = 0;
8     std::unordered_map<std::string, size_t> counts;
9     for (const auto& r : res) { // üfr jeden Hit in hits
10      n += r.matches.size(); // n + Anzahl der Matches
11      for (const auto& m : r.matches) { // over all Matches
12          if (max <= counts[r.path]++)  // Abbruchkriterium
13            break;
14        // Hits werden erstellt.
15        auto& hit = request.AppendHit()
16                           .SetPath(r.path)
```

```
17                          .SetN(r.n)
18                          .SetF(r.f)
19                          .SetAbnr(std::stoul(r.abnr))
20                          .SetSatznr(std::stoul(r.satznr))
21                          .SetCounter(i++);
22          // insert other
23                for (const auto& p: r.other) {
24                  hit.SetKeyVal(p.first, p.second);
25                }
26          // insert Matches
27          for (auto i = m.first; i != m.second; ++i)
28            hit.push_back(*i);
29        }
30      }
31      request.SetNumberOfHits(n);
32  }
```

7.1.2 reply.json

```
1  {
2      // Die antwort des servers enthaelt alle variablen der
           Anfrage.
3      "builderType" : "SimpleBuilder",
4
5       // hits ist ein array in denen die treffer stehen.
6       // jeder hit repraesentiert eine gefundene textstelle
7      "hits" :
8      [
9          // jeder hit ist ein json objekt, dass alle informationen
               ueber
10         // den treffer enthaelt.
11   {
12       // absatznummer der textstelle
13       "abnr" : 54,
14       // f idenifikation der textstelle
15       "f" : "Ts-213,iii-r",
16       // matches ist ein array, das informationen ueber die
             gematchten token enthaelt
17       "matches" :
18       [
19     {
20         // grammatical code (lexikon information des gematchten
               token)
21         "gc" : "+V+refl+tr",
22         // inflectional code (lexikon information des gematchten
               token)
23         "ic" : ":2mGi:3eGi",
24         // eindeutige interne id des token
25         "id" : 10383,
26         // offset position des token im satz
27         "pos" : 3,
28         // treetagger tag
```

```
29          "tag" : "VVFIN",
30          // der gematchte token
31          "token" : "denkt"
32      }
33        ],
34        // n identifikation der textstelle
35        "n" : "Ts-213,iii-r[8]_1",
36        // nummer im array (unnoetig?)
37        "no" : 0,
38        // pfad zum dokument, in dem die textstelle steht.
39        "path" : "/home/flo/devel/wast/wf/build/../data/Ts-213_NORM
            -tagged.xml",
40        // satznummer der textstelle
41        "satznr" : 128
42    },
43    {
44        "abnr" : 73,
45        "f" : "Ts-213,iv-r",
46        "matches" :
47        [
48      {
49          "gc" : "+V+intr+refl+tr",
50          "ic" : ":1mGc:1mGi:3mGc:3mGi:OI",
51          "id" : 7938,
52          "pos" : 14,
53          "tag" : "VVFIN",
54          "token" : "denken"
55      }
56        ],
57        "n" : "Ts-213,iv-r[8]_1",
58        "no" : 1,
59        "path" : "/home/flo/devel/wast/wf/build/../data/Ts-213_NORM
            -tagged.xml",
60        "satznr" : 179
61    },
62    ...
63      ],
64      "max" : 25,
65      // insgesamte anzahl der treffer die gefunden worden waehren,
            wenn max = 0
66      "numberOfHits" : 275,
67      // ausgabedatei (name der datei, in die diese antwort
          geschrieben wurde (wird nur von wf_server, wf_client und
          wf_display benoetigt)
68      "outputFile" : "/dev/stdout",
69      "paths" :
70      [
71    "../data/Ts-213_NORM-tagged.xml"
72      ],
73      "query" : "denken",
74      "queryType" : "UserQuery",
75      // antwortzeit des servers (wird von wf_client hinzugefuegt)
76      "responseTimeMs" : 402,
```

```
77      "skipperType" : "NoSkip",
78      "verify" : 0
79  }
```

7.1.3 RankedSearch.cpp/.hpp

```
1  #ifndef wf_RankedSearch_hpp__
2  #define wf_RankedSearch_hpp__
3
4
5  #include "SearchDocument.hpp"
6  #include "Message.hpp"
7  #include <regex>
8
9
10 namespace wf {
11 namespace dict {
12 class Dictionary;
13 }
14 struct Entry;
15 class Message;
16
17 class RankedSearch
18 {
19 public:
20   RankedSearch(Message& request,SearchDocument::Message& res,
       size_t max,const dict::Dictionary* dict_);
21   void buildRankedAnswer(Message& request,
22                          SearchDocument::Message& res,
23                          size_t max) const;
24
25
26 private:
27   double tokenDistance(std::vector<Token>::const_iterator&
       curr_it,
28                        std::vector<Token>::const_iterator &foll_it)
                           const;
29   std::unordered_map<const dict::Entry *, double> split(std::
       string query,
30                                                         int &
                                                             query_size
                                                             )
                                                             const;
31   void fillHitMap(std::unordered_map<const dict::Entry *, double>
       &query_map,
32                   std::unordered_map<const dict::Entry *, double>
                       &hit_map,
33                   const dict::Entry * &match_entry) const;
34   double fillSingleHitMap(std::unordered_map<const dict::Entry *,
       double> &query_map,
35                   int query_size,
36                   const dict::Entry * &match_entry) const;
```

```
37   void fill_unord_map(std::unordered_map<std::string,std::pair<
         double, double>> & m,
38                                                 std::string docName,
39                                                 double s_rank ) const;
40   std::string getDocumentAbsatz(std::string siglum) const;
41   const dict::Dictionary* dict_;
42 };
43 }
44 #endif /* wf_RankedSearch_hpp__ */
```

```
1  #include "RankedSearch.hpp"
2  #include "Message.hpp"
3  #include "dict/Dictionary.hpp"
4  #include "util/string.hpp"
5  #include <algorithm>
6  #include <iostream>
7
8  bool
9  sortByAbsatz(const Json::Value& a, const Json::Value& b)
10 {// For sorting Hits
11     return a["ab_rank"].asDouble() > b["ab_rank"].asDouble();
12 }
13
14
15 wf::RankedSearch::RankedSearch(Message& request,
16                                SearchDocument::Message& res,
17                                size_t max,
18                                const dict::Dictionary* dict)
19   : dict_(dict)
20 {
21   buildRankedAnswer(request, res, max);
22 };
23
24 void
25 wf::RankedSearch::buildRankedAnswer(Message& request,
26                                SearchDocument::Message& res,
27                                size_t max) const
28 {
29   std::unordered_map<std::string, std::pair<double,double>>
         absatz_ranks; // um Absatz Rank zu äzhlen
30   // map each query entry to its frequenzy
31   int query_size = 0;
32   if (res.size() == 0){
33     return; // Funktion verlassen!
34   }
35
36   std::unordered_map<const wf::dict::Entry *, double> query_map =
         split(request.oldquery, query_size); // Get Query Entrys
37   // maps each query entry to its hit frequenzy
38   std::unordered_map<const wf::dict::Entry *, double> hit_map;
39   hit_map.insert(query_map.begin(), query_map.end());
40
```

```
41  double ab_rank = 0.0;
42  for (auto& r : res) { // üfr jeden Hit in hits
43
44    for (auto &it : hit_map){  // set each value to 0
45      it.second = 0.0;
46    }
47      std::string docAbsatz = getDocumentAbsatz(r.n);
48      auto& hit = request.AppendHit()
49                      .SetPath(r.path)
50                      .SetN(r.n)
51                      .SetF(r.f)
52                      .SetAbnr(std::stoul(r.abnr))
53                      .SetAbsatzRank(ab_rank)
54                      .SetSatznr(std::stoul(r.satznr))
55                      .SetDocAbsatz(docAbsatz);
56
57      double distance= 0.0;
58      if (r.matches.size()>=2){
59        for(std::vector<Match>::size_type pos = 0; pos != r.
             matches.size()-1; pos++) { // Matches iterieren
60          auto curr_it = r.matches[pos].first;
61          auto foll_it = r.matches[pos+1].first;
62          const wf::dict::Entry* cur_match_entry = (*curr_it).
               entry(); // Dict-Entry von Match
63          fillHitMap(query_map, hit_map, cur_match_entry);
64
65          distance += tokenDistance(curr_it, foll_it);
66          hit.push_back(*curr_it); // add match
67
68          if (pos == r.matches.size()-2){ //Im Letzten durchlauf
               auch den nachfolgenden Iter adden
69            const wf::dict::Entry* foll_match_entry = (*foll_it).
                 entry();
70            fillHitMap(query_map, hit_map, foll_match_entry);
71            hit.push_back(*foll_it); // Letztes Match zu Matches
                 adden
72          }
73        }
74
75        // calculate match quality
76        bool perfect_match = true;
77        bool all_found = true;
78        double quality = 0.0;
79        for (auto it : hit_map){
80          if ((it.second < query_map[it.first]) && (it.second >
               0.0)){
81            perfect_match = false;
82            quality += it.second;
83          }else if (it.second == 0.0){
84            perfect_match = false;
85            all_found = false;
86            quality -= query_map[it.first] * (double)2; //
                 punishment
```

```
87            }else {
88              quality += it.second;
89            }
90          }
91
92
93          if (perfect_match){
94          r.s_rank = quality / (distance / (double)5.0);
95            hit.SetSatzRank(r.s_rank);
96          } else if (all_found) {
97            r.s_rank = quality / (distance/ (double)2.0);
98            hit.SetSatzRank(r.s_rank);
99          } else {
100           r.s_rank = quality / distance;
101           hit.SetSatzRank(r.s_rank);
102         }
103         fill_unord_map(absatz_ranks, docAbsatz, r.s_rank); //
              increase absatz rank
104
105       } else {
106         r.s_rank = (double)1 / (query_size*2.0); //durch die
              Query Size teilen.
107
108         hit.SetSatzRank(r.s_rank);
109         fill_unord_map(absatz_ranks, docAbsatz, r.s_rank); //
              count Absatz Rank mit
110
111         for (auto m : r.matches){
112           for (auto i = m.first; i != m.second; ++i){
113             hit.push_back(*i);
114           }
115         }
116       }
117
118         for (const auto& p: r.other) {
119           hit.SetKeyVal(p.first , p.second);
120         }
121
122   }
123   request.SetNumberOfHits(request.json_["hits"].size());
124
125
126   // Json array needs to be coppied,
127   // otherwise not sortable because sort needs
        RandomAccessIterator not Json::ValueIterator)
128   std::vector<Json::Value> copy;
129   for (auto i : request.json_["hits"]){
130     auto key = i["DocAbsatz"].asString();
131     double ab_rank = absatz_ranks[key].first / absatz_ranks[key].
          second; //durch Anzahl der Hits von dem Absatz teilen.
132     i["ab_rank"] = ab_rank;
133     copy.push_back(i);
134   }
```

```
135   std::sort(copy.begin(), copy.end(), sortByAbsatz); // Sort Hits
          by Absatz
136
137   //sortierten Vector wieder in Json::Value verwandeln
138   Json::Value sorted_hits;
139   size_t c = 0;
140   for (auto it : copy){
141     if (c <= max) { // max wird von search.js auf 200 gesetzt
142       it["no"] = static_cast<Json::UInt>(c); // Hit-Nummer
              anpassen
143       sorted_hits.append(it); // Reduce Hits to max size
144     }
145     c++;
146   }
147   // request bekommt sortierten Vector
148   request.json_["hits"] = sorted_hits;
149 }
150
151
152 //
      ////////////////////////////////////////////////////////////////////////////////

153 double
154 wf::RankedSearch::tokenDistance(std::vector<Token>::
      const_iterator &curr_it,
155                                 std::vector<Token>::const_iterator &
                                      foll_it) const
156 { // Calculate distance between the two matches
157   double distance = 0.0;
158   auto token_1 = *curr_it;
159   auto token_2 = *foll_it;
160   double pos1 = static_cast<double>(token_1.pos()); // wegen
          size_t
161   double pos2 = static_cast<double>(token_2.pos());
162   distance = pos2 -pos1;
163   return distance;
164 }
165
166 //
      ////////////////////////////////////////////////////////////////////////////////

167 std::unordered_map<const wf::dict::Entry *, double>
168 wf::RankedSearch::split(std::string query, int &query_size) const
169 {
170   std::unordered_map<const wf::dict::Entry *, double>
          query_entries;
171   std::istringstream splitter(query);
172   std::string word;
173   while(splitter >> word) {
174     query_size++;
175     std::wstring wide_str = wf::util::widen(word.c_str());
176     //std::wcout << wide_str << std::endl;
177     const wf::dict::Entry * token_entry = dict_->find(wide_str);
```

```
178     if (token_entry){
179       if (query_entries.find(token_entry) != query_entries.end())
              {
180         query_entries[token_entry] += (double)1.0;
181       } else {
182         query_entries[token_entry] = (double)1.0;
183         //std::wcout << "Query: "<< token_entry->str<< std::endl;
184       }
185     }
186   }
187   return query_entries;
188 }
189
190 //
    ////////////////////////////////////////////////////////////////////////////////
191 void
192 wf::RankedSearch::fillHitMap(std::unordered_map<const wf::dict::
    Entry *, double> &query_map,
193                      std::unordered_map<const wf::dict::Entry *,
                         double> &hit_map,
194                      const wf::dict::Entry * &match_entry) const
195 {
196
197     for (auto query_pair : query_map){
198       auto query_entry = query_pair.first;
199       size_t vPartFound = match_entry->gc.str().find(L"+V+#");
200       if (vPartFound != std::string::npos){
201         hit_map[query_entry] += (double) 3.0;
202       }
203       if (query_entry == match_entry){
204         hit_map[query_entry] += (double)1.0 / query_map[
              query_entry];
205         //std::wcout << "Literal: " << hit_map[query_entry] <<
              std::endl;
206         break;
207       } else if ((*query_entry).isLemmaOf(match_entry)) {
208         hit_map[query_entry] += (double)0.9 / query_map[
              query_entry];
209         //std::wcout << "OTHER LEMMA!" <<hit_map[query_entry] <<
              std::endl;
210         break;
211       } else if ((*match_entry).isInverseLemmaOf(query_entry)){
212         //std::wcout << "inverse" << std::endl;
213         hit_map[query_entry] += (double)0.9 / query_map[
              query_entry];
214         break;
215       }
216       else {
217         //std::wcout << "no match" << std::endl;
218         continue;
219       }
220     }
```

```
221 }
222
223 void
224 wf::RankedSearch::fill_unord_map(std::unordered_map<std::string,
      std::pair<double, double>> & m,
225                       std::string docAbsatz,
226                       double s_rank) const
227 {
228   auto it = m.find( docAbsatz );
229   if ( it == m.end() ) {
230     auto value_pair = std::make_pair(s_rank,1.0); // 1.0 Anzahl
        des Absatzes
231     m.insert(std::make_pair(docAbsatz, value_pair));
232   }
233   else {
234    m[docAbsatz].first += s_rank;
235    m[docAbsatz].second += 1.0; // öErhhe anzahl der äAbstze
236   }
237 }
238
239 std::string
240 wf::RankedSearch::getDocumentAbsatz(std::string siglum) const{
241   std::smatch match;
242   std::regex rgx("(.*)_");
243   if (std::regex_search(siglum, match, rgx)){
244     return match[1];
245   } else {
246     return siglum;
247   }
248 }
```

7.2 Suchergebnisse

Suchergebnisse zu “Gedanken Welt”

```
1  {{
2    "DocAbsatz": "Ts-212,1028[1]",
3    "ab_rank": 3.3333333333333335,
4    "s_rank": 3.3333333333333335,
5    "matches": [
6      {"token": ["Welt", "Gedanken"],
7        "pos": [ 7, 10 ]
8      }]
9  }, {
10   "DocAbsatz": "Ts-211,399[2]",
11   "ab_rank": 3.3333333333333335,
12   "s_rank": 3.3333333333333335,
13   "matches": [
14     {"token": ["Welt", "Gedanken"],
15       "pos": [ 7, 10]
16     }]
17 }, {
```

```
  "DocAbsatz": "Ts-213,370r[2]et369v[2]",
  "ab_rank": 2,
  "s_rank": 2,
  "matches": [
    {"token": ["Welt", "Gedanken"],
      "pos": [ 8, 13 ]
    }]
}, {
  "DocAbsatz": "Ts-212,1062[1]",
  "ab_rank": 0.9570512820512821,
  "s_rank": 0,
  "matches": [
    {"token": ["Gedanken", "Gedanken"],
      "pos": [ 9, 28 ]
    }]
},{
  "DocAbsatz": "Ts-212,1062[1]",
  "ab_rank": 0.9570512820512821,
  "s_rank": 0.25,
  "matches": [
    {"token": ["Gedanken"],
      "pos": [ 4 ]
    }]
},{
  "DocAbsatz": "Ts-212,1062[1]",
  "ab_rank": 0.9570512820512821,
  "s_rank": 0.25,
  "matches": [
    { "token": [ "Gedanken" ],
      "pos": [53]
    } ]
}, {
  "DocAbsatz": "Ts-212,1062[1]",
  "ab_rank": 0.9570512820512821,
  "s_rank": -0.007692307692307699,
  "matches": [
    { "token": ["Gedanken","Gedanke"],
      "pos": [17,30]
    }]
},{
  "DocAbsatz": "Ts-212,1062[1]",
  "ab_rank": 0.9570512820512821,
  "s_rank": 5,
  "matches": [
    {"token": [ "Welt","Gedanken"],
      "pos": [ 12,14 ]
    }]
}, {
  "DocAbsatz": "Ts-212,1062[1]",
  "ab_rank": 0.9570512820512821,
  "s_rank": 0.25,
  "matches": [
    { "token": ["Gedanken"],
```

```
71        "pos": [5]
72      }]
73  }
```

Suchergebnisse zu “man weiß”

```
1   {
2     "DocAbsatz": "Ts-213,149v[2]",
3     "ab_rank": 10,
4     "s_rank": 10,
5     "matches": [
6       { "token": [ "ßwei","man"],
7         "pos": [1, 2]
8       } ]
9   },{
10    "DocAbsatz": "Ms-142,105[2]et106[1]",
11    "ab_rank": 10,
12    "s_rank": 10,
13    "matches": [
14      {"token": ["man","ßwei"],
15        "pos": [0, 1]
16      } ]
17  },{
18    "DocAbsatz": "Ts-211,656[4]et657[1]",
19    "ab_rank": 10,
20    "s_rank": 10,
21    "matches": [
22      {"token": ["ßwei", "man"],
23        "pos": [ 8, 9]
24      }]
25  },{
26    "DocAbsatz": "Ts-212,1894[1]et1895[1]",
27    "ab_rank": 10,
28    "s_rank": 10,
29    "matches": [
30      {"token": ["ßwei","man"],
31        "pos": [8,9 ]
32      } ]
33  }
34  {
35    "DocAbsatz": "Ts-211,498[2]",
36    "ab_rank": 5.58,
37    "s_rank": 1.16,
38    "matches": [
39      { "token": ["man","wissen","man"],
40        "pos": [2,4,7]
41      } ]
42  },{
43    "DocAbsatz": "Ts-213,55r[1]",
44    "ab_rank": 5.58,
45    "s_rank": 10,
46    "matches": [
47      {"token": ["ßwei","man"],
```

```
        "pos": [3,4]
      } ]
},{
  "DocAbsatz": "Ts-213,55r[1]",
  "ab_rank": 5.58,
  "s_rank": 1.16,
  "matches": [
    {"token": ["man","wissen","man"],
      "pos": [ 2, 4, 7]
    } ]
},{
  "DocAbsatz": "Ts-211,498[2]",
  "ab_rank": 5.58,
  "s_rank": 10,
  "matches": [
    {"token": ["ßwei","man"],
      "pos": [3,4]
    }]
}{
  "DocAbsatz": "Ts-212,187[2]",
  "ab_rank": 5.58,
  "s_rank": 1.16,
  "matches": [
    {"token": ["man","wissen","man"],
      "pos": [2,4,7]
    }]
},{
  "DocAbsatz": "Ts-212,187[2]",
  "ab_rank": 5.58,
  "s_rank": 10,
  "matches": [
    {"token": ["ßwei","man"],
      "pos": [3,4]
    }]
}
```


Suchergebnisse zu "Haus Stadt"


```
{
  "DocAbsatz": "Ts-213,727r[2]et728r[1]",
  "ab_rank": 3.3333333333333335,
  "s_rank": 3.3333333333333335,
  "matches": [
    {"token": ["Haus","Stadt"],
      "pos": [22,25]
    }]
},{
  "DocAbsatz": "Ts-211,630[2]et631[1]",
  "ab_rank": 3.3333333333333335,
  "s_rank": 3.3333333333333335,
  "matches": [
    {"token": ["Haus","Stadt"],
      "pos": [23,26]
```

```
16      }]
17  },{
18    "DocAbsatz": "Ts-212,1766[1]et1767[1]",
19    "ab_rank": 3.3333333333333335,
20    "s_rank": 3.3333333333333335,
21    "matches": [
22      {"token": ["Haus","Stadt"],
23        "pos": [22,25]
24      }]
25  },{
26    "DocAbsatz": "Ts-213,185r[8]et186r[1]",
27    "ab_rank": 2.5,
28    "s_rank": 2.5,
29    "matches": [
30      {"token": ["Haus","Stadt"],
31        "pos": [26,30]
32      }]
33  },{
34    "DocAbsatz": "Ts-212,571[2]",
35    "ab_rank": 2.5,
36    "s_rank": 2.5,
37    "matches": [
38      {"token": ["Haus","Stadt"],
39        "pos": [24,28]
40      }]
41  },{
42    "DocAbsatz": "Ts-211,297[3]",
43    "ab_rank": 2.5,
44    "s_rank": 2.5,
45    "matches": [
46      {"token": ["Haus","Stadt"],
47        "pos": [26,30]
48      }]
49  },{
50    "DocAbsatz": "Ms-142,12[2]et13[1]",
51    "ab_rank": 0.44625,
52    "s_rank": 0.58,
53    "matches": [
54      {"token": ["äHusern","Stadt","Stadt"],
55        "pos": [4,11,14]
56      }]
57  },{
58    "DocAbsatz": "Ms-142,12[2]et13[1]",
59    "ab_rank": 0.44625,
60    "s_rank": 0.3125,
61    "matches": [
62      {"token": ["Stadt","äHusern","äHusern","äHusern","äHusern","
          äHusern"],
63        "pos": [8,35,50,66,69,96]
64      }]
65  },{
66    "DocAbsatz": "Ts-211,579[2]et580[1]",
67    "ab_rank": 0.25,
```

```
68    "s_rank": 0.25,
69    "matches": [
70      {"token": ["Haus"],
71        "pos": [6]
72      }]
73  },{
74    "DocAbsatz": "Ts-211,580[2]et581[1]",
75    "ab_rank": 0.25,
76    "s_rank": 0.25,
77    "matches": [
78      {"token": ["Haus"],
79        "pos": [7]
80      }]
81  }
```

7.3 READMEs aus Gitlab

7.3.1 Sphinx

```
1 # Prerequisites:
2 * Sphinx: `sudo pip3 install -U Sphinx`
3 * recommonmark `sudo pip3 install -U recommonmark`
4 * Sphinx-Markdown-Tables: `pip install sphinx-markdown-tables`
5
6 # Create HTML-Page:
7
8 1. `cd PATH_TO_FOLDER/`
9 2. `make html` (Warnings can be ignored for now.)
10 3. `firefox build/html/index.html`
```

7.3.2 wf

```
1 There are two simple tools for wittfind:
2
3  * the wf tool searches for queries and gives out a list of
       matches.
4  * the wf_display tool uses the input of wf to display the
       matches in the original file.
5  * the wf_server tool starts a server listening on a unix domain
       socket.
6  * the wf_client tool queries a server that waits on a unix
       domain socket.
7  * all tools know the -h [--help] option.
8
9 All tools are built in the build/bin folder
10 (if you use the custom Makfile, they are copied directly into
     your current directory).
11
12     $ ./wf --help
13     $ ./wf -d dictionary -f input -q "query" -m max -o outfile
```

```
14      $ ./wf -d dictionary -f input -Q query-file -m max -o outfile
15      $ ./wf -L dicdictionary -f input -q "query"
16 max specifies the maximal number of hits shown. Default is 25, if
       0 all matches are shown.
17
18      $ ./wf_display --help
19      $ ./wf_display [hits]
20      $ ./wf -d data/witt_WAB_dela.txt -f data/witt_input_tagged.
          xml -q /\d+/ | ./wf_display
21      $ ./wf -v -d data/witt_WAB_dela.txt -f data/witt_input_tagged
          .xml -q /\d+/ | ./wf_display -v
22
23 creating wf_display xml output for the web front end
24 (-t B marks hits with <B>hit</B>, -r hits puts the results in <
      hits>...</hits> tags):
25      $ ./wf_display -t B -r hits pfad/zur/serverdatei.xml
26
27 starting the server:
28      $ cd wf/build/
29      $ ./bin/wf_server --dictionary ../data/witt_WAB_dela_XI.txt
          --files ../data/files.txt
30
31 using the client:
32
33      $ ./bin/wf_client --query '[VVFIN] & denken' -f ../data/Ts
          -213_NORM-tagged.xml \
34     --max 10 --threads 2
35
36 ## Query syntax
37  * path/to/graph.json(arg1, arg2, ..., argn) loads a subgraph
       from path/to/graph.json
38    and replaces the arguments $1$, $2$, ... $n$ with arg1, arg2,
          ..., argn.
39    It is not possible to concatenate subgraph expressions with
          other expressions in a node.
40  * token matches any token that is either 'token' or has 'token'
       as its lemma.
41  * "token" or 'token' matches any token that is equal to token.
42  * \<GC\> matches any token with the grammtical code 'GC'.
43  * [TAG] matches any token with an annotation equal to 'TAG'
44  * /regex/ matches any token that matches the regular expression
       'regex'.
45    Note: if you want the regex to match the whole token you have
          to use '/^regex$/'.
46  * /regex/i matches any token that matches the regular expression
        'regex' ignoring case.
47  * [/regex/] matches all token whose tag matches the regular
       expression 'regex'.
48  * You can prefix any query expression with '!' to prohibt the
       higlighting of this particular match.
49    E.g. "/\w+/ !\<PUNCT\> /\w+/" matches words, followed by
          punctuation and another word,
```

```
50    but the punctuation is not highlighted as a match (the two
          words are, though):
51    [[[foo]]] , [[[bar]]]
52  * You use boolean operators '(', ')', '&', '|' or '~' in a node
       to form complex expressions:
53      * '/en$/ & [N]' matches token that end on 'en' and have the
            tag N.
54      * '/en$/ | [N]' matches token that end on 'en' or have the
            tag N.
55      * '~ /en$/' matches token that _don't_ end with 'en'.
56      * use bracets to form more complex expressions.
57      * the parser for complex expressions is not finished yet. You
            need to use explicit whitespace
58        to seperate operators and expressions: '(/a/|/b/)&~/c/' is
            invalid.
59        Use: '( /a/ | /b/ ) & ~ /c/'.
60  * There are some unitex special expressions: \<MAJ\>, \<MIN\>,
       \<MOT\>, \<PRE\> and \<NB\>
61    wich all map to speical regex pattern.
62  * It is possible to use '?' to make the previous expression
       optional.
63    The query 'a b? c' matches the token 'a b c' or 'a c'.
64  * It is possible to specify a range expression '{f, t}' to match
         the previous match f to t times.
65    The query 'x {3, 5}' matches at least 3 xs up to 5 xs.
66    The Expression '{x}' is shorthand for '{x, x}'.
67  * You can append '+' to an expression to make it match one or
       more times.
68    The Query a <MOT>+ b matches any expression 'a' followed by
          one or more words and a 'b'.
69  * You can append '*' to an expression to make it match zero or
       more times.
```

Literaturverzeichnis

[1] CPP-Reference. Algorithm - std::sort. Verfügbar unter `https://en.cppreference.com/w/cpp/algorithm/sort` [Accessed: 01-Jun-2019].

[2] CPP-Reference. C++ named requirements: Compare. Verfügbar unter `https://en.cppreference.com/w/cpp/named_req/Compare` [Accessed: 01-Jun-2019].

[3] Dr. Max Hadersbeck, Daniel Bruder. WAST - Wittgenstein Advanced Search Tools Dokumentation. CIS, LMU, München, 2014.

[4] Bharat Krishna. Ranking search results by reranking the results based on local interconnectivity. U.S. Patent No. 6.526.440., 25 Feb. 2003.

[5] Christopher D. Manning, Prabhakar Raghavan, Hinrich Schütze.Introduction to information retrieval. Cambridge University Press. 2008.

[6] Friedhelm Padberg, Sebastian Wartha. Didaktik der Bruchrechnung. Herder, Freiburg, 1978.

[7] Juan Ramos. Using tf-idf to determine word relevance in document queries. In: Proceedings of the first instructional conference on machine learning. Vol. 242. 2003, [S. 133-142].

[8] Prof. Dr. Klaus U. Schulz. Formale Sprachen und Automaten. CIS, LMU, München, 2011

[9] Sphinx-Doc. Projects using Sphinx. verfügbar unter `http://www.sphinx-doc.org/en/master/examples.html` [Accessed: 01-Jun-2019].

[10] Bjarne Stroustrup. The C++ programming language, Pearson Education India, 2000.

[11] Wikipedia. Sphinx (Software). Verfügbar unter `https://de.wikipedia.org/wiki/Sphinx_(Software)` [Accessed: 01-Jun-2019].

Abbildungsverzeichnis

Inhalt des beigelegten USB-Sticks

Folgende Dateien beinhalten neuen Code:

- RankedSearch.cpp/.hpp
- Searcher.cpp/.hpp
- Message.cpp/.hpp
- SearchDocument.cpp/.hpp
- Server_test.cpp
- CMakeLists.txt

Folgende Dateien beinhalten nur neue Kommentare:

- Search.hpp
- Dictionary.cpp./.hpp

Sonstige Dateien:

- BA.pdf