Army ANT: A Workbench for Innovation in Entity-Oriented Search

External Option: Scientific Activities – TREC Open Search

MAP-i 2016/2017

José Devezas
joseluisdevezas@gmail.com

June 9, 2017

Contents

Abstract

This report has been submitted in partial fulfillment of the requirements for the approval of the External Option curricular unit, of the MAP-i Doctoral Programme in Computer Science, 2016/2017 edition. In this document, we present the scientific output regarding the work developed in preparation for the Open Search track of the Text REtrieval Conference (TREC), in the context of the Scientific Activities curricular option. We being by presenting an overview of how TREC works. Search engine evaluation in TREC has been historically based on a set of fixed topics, accompanied by relevance judgments provided by human assessors. The Open Search track changed this approach by introducing the concept of a Living Labs, where results can be evaluated by real users of real web search engines. The calendar for the 2017's edition of this track hasn't been released as of May 2017. Accordingly, we present the system we prepared to enable the participation in the TREC Open Search track, when it becomes available.

Army ANT is a workbench we developed to support innovation in entity-oriented search, which also makes it an essential tool for this doctoral work. While it provides an adequate level of freedom for information retrieval researchers to build their own search engine implementations, it also provides reusable code, such as a basic analyzer for tokenization and stopword removal, as well as some structure to enable the integration of different engines. Each engine should provide documentation on its ranking functions and indexed datasets, as well as the values for the score components of individual documents. Army ANT then provides a parallel coordinates visualization to support the understanding of the ranking function through its individual components, as an aid to explain the final score of each document.

We implemented two graph-based document representation and retrieval models that we integrated with Army ANT. First, we studied and explored the graph-of-word model. This was originally implemented using an inverted file, replacing the term frequency by a term weight that was computed based on the graph-of-word for each individual document. Our implementation, however, was done directly on a graph database, and the ranking function was implemented using the Gremlin domain-specific language (DSL). While this severely impacted efficiency, both for indexing and retrieval, it provided a purely graph-based approach that enabled a focused and clear method to explore the relations among terms, entities, and terms and entities. As one of the goals of this doctoral thesis is to prove that several retrieval tasks can be unified in a single model, in particular through a graph data structure, we felt this was a fundamental step, in spite of expected performance issues. While the graph-of-word only contained a single dimension of nodes representing terms, we also proposed an improvement and extension of this model, which we called graph-of-entity. The graph-of-entity slightly redefines the edges between term nodes (it does not use sliding windows) and it also includes entity nodes. Nodes are connected through edges that model all the available structured and unstructured information as a unified document representation. We then integrate two essential tasks of entity-oriented search through this model — query entity linking and entity ranking — and show that query entity linking can directly contribute to the ranking process instead of being considered a separate step. The intuition was that the uncertainty of query entity linking should seamlessly contribute to the weighting scheme as a regular feature.

Given the unexpected delay of the TREC Open Search track, we were required to implement our own evaluation benchmark, based on available datasets. In particular, we used the Wikipedia Relation Extraction Data v1.0, which provides textual content about an entity, along with labeled relations to other entities. This dataset enabled us to build the first implementations of the graph-of-word and graph-of-entity models. However, since it did not provide any relevance judgments, we also indexed a subset of the INEX 2009 Wikipedia Collection, which provided a two-fold contribution to this work. On one hand, it enabled us to test the indexing process of a dataset with a different format and, on the other hand, it provided a list of topics, assessed by real users, that could be used to evaluate the models, calculating metrics like the mean average precision (MAP) or the normalized discounted cumulative gain for the top-p results (NDCG@p).

We close with some final remarks on Army ANT and the two graph-based document representation and retrieval models that we implemented. Given the lack of enough evidence, we could not conclude whether there is an improved effectiveness for the graph-of-entity over the graph-of-word model. With a little tuning, however, our platform should be able to successfully index the complete INEX 2009 Collection, enabling us to assess effectiveness and further extend the range of unified entity-oriented search subtasks supported by the graph-of-entity.

Chapter 1

Introduction

In this chapter, we introduce the Text REtrieval Conference (TREC), covering some of its most relevant tracks on entity-oriented and semantic search, as well as presenting the Open Search track, for which we are preparing through the work we describe in this report. We also describe the methodology used to structure the process of innovation in entity-oriented search and, in particular, how we will take advantage of the Living Labs for the second Academic Search Edition of the TREC Open Search track to assess the effectiveness of two graph-based models.

Text REtrieval Conference

The Text REtrieval Conference (TREC) began in 1992 and has ever since brought together the information retrieval (IR) community to participate in several research tracks. Each research track represents a different open challenge in the area of IR — tracks can be discontinued when a problem has been solved or interest has faded, and new tracks are frequently created to better represent new relevant challenges in the area. TREC participants are expected to choose one or multiple tracks, each providing specific resources (e.g., document collections, relevance judgments, APIs, etc.), in order to develop a search engine that will be evaluated on a common framework. The event is organized as a competition rather than a typical conference. In the past, FEUP has already participated in TREC multiple times. In particular, during the last Blog Track, in 2010, we explored the effects of graph-based query-independent features for blog retrieval, comparing the indegree with the h-index, as two link analysis metrics —– we established an analogy to bibliometrics, where we considered blogs as scientists and posts as publications (Devezas, Nunes, and Ribeiro [2010]). We found that the indegree actually decreased performance, while the h-index was a good score component to improve blog retrieval.

As the result of a TREC participation, a research paper must be submitted to publish in a NIST Special Publication dedicated to TREC, describing the approach taken and the obtained results. The focus is on using traditional IR metrics, such as mean average precision (MAP) or normalized discounted cumulative gain (NDCG@p), to measure the quality of different models or different parameter values based on human relevance judgments. Most tracks, including the Open Search Track (Balog et al. [2016]), also provide an overview on the competition, where the quality of the participants' search engines is compared. This works as a state-of-the-art assessment of a particular information retrieval task.

Knowledge Base Acceleration Knowledge Base Acceleration (KBA) ran consecutively from 2012 to 2014 and was succeeded by the Dynamic Domain Track in 2015, which also ran in 2016 and is running again in 2017. KBA describes their mission as follows[1]: "Given a rich dossier on a subject, filter a stream of documents to accelerate users filling in knowledge gaps.". This track tackled the challenge of increasing the speed at which a news article is cited in a knowledge base from the moment of its publication. In 2014, the median number of days a news article waited to be cited in Wikipedia was 356 days (nearly a year!)[2]. The KBA track dealt with this issue by focusing on improving entity-oriented filtering of large

[1] `http://trec-kba.org/`
[2] `http://trec-kba.org/data/2014-11-19-TREC-KBA-track-overview.pptx`

Table 1.1: Overview of semantic and entity-oriented research tracks from TREC.

Track	Datasets	Mission
Knowledge Base Acceleration Track[1]	TREC KBA Stream Corpora 2012-2014[3]	Improve the retrieval of documents about a particular entity from a stream, in order to accelerate manual knowledge base population.
Entity Track	ClueWeb09[5], BTC-2009[6], BTC-2010[6]	Improve the retrieval of related entities based on one or multiple entities.
Question Answering Track[8]	TIPSTER[10], TREC disks[9], MSNSearch logs, AskJeeves logs, The AQUAINT Corpus of English News Text[11]	Improve the retrieval of document passages capable of directly answering user questions.
Live QA Track[12]	Yahoo! Answers[13]	Similar to the Question Answering Track, but questions are received directly from a socket, as a stream, in real-time.
Open Search Track[14]	CiteSeerX[15], Microsoft Academic Search[16]	The Academic Search Edition consists of improving literature retrieval for a given a keyword query, which might include entities (e.g., authors).

streams, in order to help human curators fill in the knowledge gaps more quickly. Tasks were based on the TREC KBA Stream Corpora 2012-2014[3]

Entity Track The Entity Track[4] ran consecutively from 2009 to 2011 and it consisted of two main tasks. The first task, Related Entity Finding (REF), was based on the ClueWeb09[5] dataset, as well as the Billion Triples Challenge datasets (BTC-2009[6] and BTC-2010[7]). The second task, Entity List Completion (ELC), was also based on the BTC-2009 and BTC-2010 Linked Open Data. The overall problem tackled by the Entity Track consisted of taking one (REF) or several (ELC) entities as a query and returning a ranked list with the best related entities.

Question Answering Track Question Answering Track[8] is one of the longest running tracks organized by TREC, starting on 1999 and running until 2007, based on several static datasets shared by other tracks, such as TREC disks 4&5[9], accompanied by a set of test questions, either manually created or taken, for instance, from search logs donated by Microsoft or AOL. The track was inactive for eight years, until 2015, when it was revived as the Live QA Track. The Live QA Track also ran in 2016 and is running again in 2017. The Live QA track was different from its precursor in the sense that it is required to find answer to questions submitted to Yahoo Answers and pushed to participants as a data stream.

Table 1.1 presents a summary on the most relevant research tracks, their datasets and mission, in the context of semantic and entity-oriented search. We also included the Open Search track since, in the particular context of this doctoral work, we will take advantage of an implicit presence of entities, namely the authors, within academic publications, to compare the impact of indexing only text versus indexing combined data (i.e., text and knowledge).

Open Search Track

We will take this opportunity to participate in TREC 2017 Open Search track with the main goal of understanding and creating a working implementation of the graph-of-word model (Rousseau and Vazirgiannis [2013]), described in Section 3.2. As a secondary goal, we aim to explore novel ways of extending this graph-based model with information such as entities or relations, while linking with the indexed text within the same data structure. We implement such approach with the graph-of-entity

[3]http://s3.amazonaws.com/aws-publicdatasets/trec/kba/index.html
[4]https://web.archive.org/web/20110811014305/http://ilps.science.uva.nl/trec-entity/
[5]https://lemurproject.org/clueweb09/
[6]https://km.aifb.kit.edu/projects/btc-2009/
[7]https://km.aifb.kit.edu/projects/btc-2010/
[8]http://trec.nist.gov/data/qamain.html
[9]http://trec.nist.gov/data/docs_eng.html

model, described in Section 3.3. The TREC 2017 participation will provide a benchmark for evaluation that will support the remaining work of this thesis as we improve the indexing and ranking strategies of our graph-based entity-oriented search framework. The event will continue beyond the ending of the semester and, therefore, in this report, we only present the preparation work we have done for the Open Search track, running in TREC 2017, describing how the graph-based models that we plan to evaluate with the academic data provided by the Living Labs API[17] for the track.

This year, the Open Search Track is partly organized by Krisztian Balog[18], a well-known IR scientist working on entity-oriented search. This task will follow last year's format, where participants are given access to an API that provides them with documents from well-known search engines, like CiteSeerX or Microsoft Academic Search. These documents must be indexed, using whatever strategy the participant chooses. Then, a list of frequent queries is provided to the participants, who must run these queries through their search engines and submit the results to the Living Labs platform. This infrastructure is linked with the real-world search engines that initially provided the documents. The search engines dedicate a small part of their traffic to evaluating runs from participants, whenever a real user issues one of the supported queries, participant's results are interpolated with the search engine's results and shown to the final user. Feedback is sent back to the participant with information on which results (theirs or the search engine's) were clicked. This implicit feedback can then be used to evaluate the system or even to train for a learning to rank approach, during the train stage. Two other evaluation stages exist before the final evaluation metrics can be computed.

Innovating Entity-Oriented Search

When, in 1990[19], Alan Emtage created the first search engine, Archie[20], search was still heavily based on keyword queries, as inspired by the search potential of back-of-the-book indexing. However, as the web evolved, so did people's information needs. Queries changed from simple topic keywords to more complex entity-oriented queries. Bautin and Skiena (2007) found that nearly 87% of all queries contained entities, based on the analysis of 36 million queries released by AOL[21].

The word entity comes from the Medieval Latin *entitas*, which means *being*. Consistently, entity is defined in the Oxford English dictionary as “a thing with a distinct and independent existence” and in the English Wikipedia as “something that exists as itself, as a subject or as an object, actually or potentially, concretely or abstractly, physically or not”.

Traditionally, in information retrieval, we define document as a kind of material of an unstructured nature, usually consisting of text (Manning, Raghavan, Schütze, et al. [2008]). According to the definition of entity, we might therefore consider any document to be an entity, however, most systems that index combined data, usually distinguish between text and knowledge (Bast, Buchhold, Haussmann, et al. [2016]). We redefine the concept of document, in the context of entity-oriented search, as having three main components: a unique identifier, an optional textual block, consisting of one or several fields, and an optional knowledge block, consisting of triples representing statements directly associated with the concept or concepts illustrated in the document.

Methodology

In order to do any research on entity-oriented search, we must first assume we have access to a collection of combined data (text and knowledge) and we must then define a specific search task over this collection. For instance, we might want to use a keyword query to retrieve text, in which case we might take advantage of any knowledge associated with the text, in order to rank documents based on links to entities (or relations) in the query. Or we might want to, instead, retrieve entities, either by name, or through context, as obtained from directly associated text or from indirectly associated text from any of the linked entities.

While obtaining a dataset is an essential step to start developing a document representation and retrieval model, it is useless without a way to assess the impact, particularly in effectiveness (as efficiency

[17]`http://doc.trec-open-search.org/en/latest/api-participant.html`
[18]`http://krisztianbalog.com/`
[19]`http://www.searchenginehistory.com/`
[20]`http://archie.icm.edu.pl/archie-adv_eng.html`
[21]`https://techcrunch.com/2006/08/06/aol-proudly-releases-massive-amounts-of-user-search-data/`

can usually be measured with time). In order to evaluate a retrieval model, we frequently use a ground truth made of a selection of topics (i.e., queries, usually with additional information, such as a narrative describing the information need) and the respective relevance judgments made by humans and usually represented through a 0 to 3 grading system. Based on the ground truth, we can them compare two different retrieval models, either representing a simple parameter tune or a completely different approach.

In order to support research work in entity-oriented search, we decided to build a workbench platform, called Army ANT, that provides a basic programmatic interface to any search engine back-end, as long as it implements `search()` and `index()` functions, where the `index()` function iterates over a reader for a particular collection, returning documents that can either contain: (*i*) solely text, (*ii*) solely triples, or (*iii*) a combination of text and triples. Using such an approach, enabled, for instance, the comparison of a text-based retrieval model with a retrieval model for combined data, potentiating the measurement of the impact of entity-awareness in search engines, which will be our main goal for the TREC 2017 Open Search track. In Chapter 2, we will provide further detail on the Army ANT workbench, namely regarding the evaluation module used to compare different engines.

Graph-Based Approach

Over the years, in information retrieval, the inverted file has been the uncontested winner in efficiency, as an index data structure inspired by back-of-the-book indexing. Zobel and Moffat (2006) present a survey on index storage, index construction and query evaluation, where they indicate two alternative index data structures and a retrieval model that were less efficient, when compared to the inverted index and state-of-the-art weighting functions, such as TF-IDF or BM25. In particular, regarding index data structures, they have listed suffix arrays (Manber and Myers [1990]) and signature files (Faloutsos and Christodoulakis [1984]) as two competing alternatives to the inverted file, that were also proposed for document indexing and retrieval. However, research over the years (Zobel and Moffat [2006]; Zobel, Moffat, and Ramamohanarao [1998]) has shown that the inverted file achieves higher compression and therefore higher efficiency. Furthermore, for the particular case of suffix arrays, there is no equivalent of ranked querying. Regarding retrieval models, cluster retrieval was also studied (Voorhees [1986]) and, when compared to inverted file approaches, it was also found to be less efficient.

This doctoral work begins an inglorious but extremely important journey through one of those alternative paths, where the goal is to explore the graph as an index data structure. This can be done either conceptually, like Rousseau and Vazirgiannis (2013), who, as we will describe in Section 3.2, used a graph to represent a text document and compute a term weight (analogous and replacing of the term frequency) that was then stored in an inverted index. This can also be done concretely by implementing a graph-based index data structure in disk and using it to issue queries and retrieve documents in real time. As we will show further along this document, in this particular case we followed the second path, implementing the two graph-based retrieval models using a graph database and incurring in (rather expected) efficiency issues. The main goal, however, was to explore and assess the graph data structure as a good (or bad) index data structure for a scenario where there are multiple heterogeneous information sources that, together, should contribute to an improved ranking of the results.

We propose that the graph would be a particularly good data structure to unify entity-oriented retrieval tasks, such as query entity linking and document ranking. There has already been work on using graphs to combine text and entities (Moro, Raganato, and Navigli [2014]), as well as on unifying machine learning models, in the quest to find a general-purpose learning algorithm, or a “master algorithm” (Domingos [2015]). In this work, we focus on unifying all information sources through a graph — as opposed to having an inverted file for full-text search and a separate triple store for inference and knowledge retrieval —, but also to integrate tasks from information extraction and information retrieval in a way that uncertainty is propagated through a pipeline in a probabilistic manner, obtaining the best possible ranked list of results without taking any deterministic assumption at a prior step.

Specifically, in this work, we show how we can build a graph of terms and entities, which we call the graph-of-entity (Section 3.3), as a logical next step to the graph-of-word (Rousseau and Vazirgiannis [2013]), used to seamlessly index text and knowledge. In this model, we implement a ranking function that is inclusive of the query entity linking process and, in fact, takes that signal into consideration when computing the final score of a document or entity, instead of using that process as a separate step where only the best linking is to be considered.

Chapter 2

Army ANT

Army ANT is a Python-based software package developed as a workbench for innovation in entity-oriented search. The goal is to make it easier for researchers, teachers or students to experiment with different search engine implementations, easily switching from between retrieval models (engines) to explore differences in results and to learn about the datasets and the scoring function, with the aid of visual tools to explore individual values of score components per document or entity. Besides centralization and exploration, it is also important, for innovation, to evaluate different retrieval models over the same queries and given a ground truth. Army ANT also provides a framework to easily implement evaluation schemes, where evaluation tasks can be queued by uploading a topics file with queries and an assessments file with relevance judgments usually given by human evaluators.

In the following sections, we will describe the Army ANT workbench from an engineering perspective. First, we will provide details on the system architecture, describing how it can be used by IR researchers or students to test and evaluate their engines on different datasets. Next, we provide a user's manual both for the command line interface and for the web interface, in order to illustrate how the workbench should be used for research or even for teaching IR.

System Architecture

A typical workflow for Army ANT begins with the implementation or reusing of a reader for a particular dataset. A reader is essentially an iterator of documents (instances of `army_ant.reader.Document`). Each `Document` contains four main fields: `doc_id` (a number or string that uniquely identifies the document), `text` (a string representing the textual block of the document), `triples` (a list of string tuples representing the knowledge block of the document, where each tuple has a subject, a predicate and an object, and the subject is usually a representation of the main concept or entity covered within the document), and `metadata` (a map of document properties). During the indexing process, the index also acts as an iterator of documents. Documents yielded by the index can be different in content from documents yielded by the reader, for example extending the knowledge block with additional entities and relations extracted from the textual block. The metadata for these documents is then stored in the database. Figure 2.1 illustrates the main dispatchable actions from the command line (index, search and server), along with the two main HTTP requests handled by the server (GET /search and POST /evaluation). Most actions, with the exception of the evaluation, involve contacting the index to either index or search documents, as well as the database to either store or retrieve the metadata for a set of documents. The evaluation is based on the queuing of tasks that batch retrieve results for a set of queries, which are then compared with a ground truth and assessed using metrics like the mean average precision (MAP) or the normalized discounted cumulative gain for the top-p results (NDCG@p).

Next, we will go through each module, describing existing implementations, as well as the necessary steps to implement new instances of `Reader`, `Index`, `Database` or `Evaluator`. We will also present an overview on the `CommandLineInterface` and `Server` modules, although we do not anticipate any extensions to be developed for these modules, as it would tamper with the normal operation of the workbench.

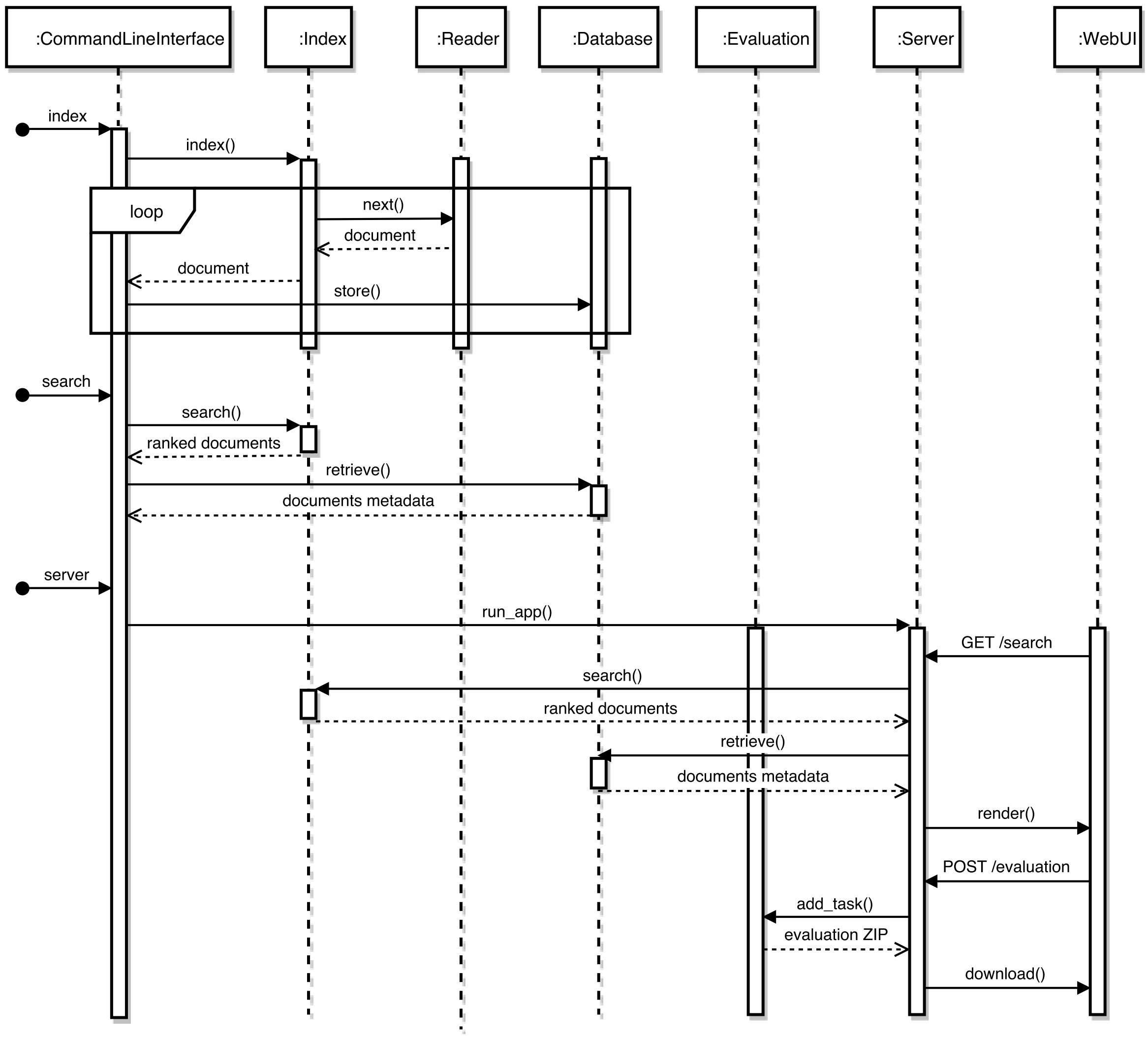


Figure 2.1: Army ANT sequence diagram for index, search and server actions.

Command Line Interface

The command line interface (CLI), implemented using Python Fire[1], provides an interface to index, search, launch the web server, and access any available extras. Python Fire automatically transforms the methods of a class into application commands and the parameters of the methods into individual command line arguments for a command. Listing 2.1 shows the usage for all commands implemented in Army ANT's CLI. As we can see, a reader is selected through the `--source-reader` argument and, since it is only used within the index command, there is no command to only read a dataset, outside of this context. There is also a `--source-path` argument to select the path of the dataset to be indexed. Generally, we use the `location` designation instead of `path` or `hostname`, but we haven't yet converted the `Reader` class to respect this convention. This is justifiable given the general aspect of Army ANT as a workbench, where a reader should be able to either iterate over documents in a database, in a compressed file or in a directory. Or an index could be accessed through a web service like Apache Solr[2] or, in our case, Apache TinkerPop Gremlin Server[3], just as easily as a disk interface to an inverted file. In fact, we have already applied this convention to the index, search and fetch-wikipedia-images commands, accepting `<hostname>` or `<hostname>:<port>` as the index location for our particular

[1]`https://github.com/google/python-fire`
[2]`http://lucene.apache.org/solr/`
[3]`https://tinkerpop.apache.org/`

Listing 2.1: Usage help for all Army ANT commands.

```
Usage: ./army-ant.py index SOURCE_PATH SOURCE_READER [INDEX_LOCATION] [INDEX_TYPE] [DB_LOCATION
    ] [DB_NAME] [DB_TYPE] [LIMIT]
       ./army-ant.py index --source-path SOURCE_PATH --source-reader SOURCE_READER [--index-
           location INDEX_LOCATION] [--index-type INDEX_TYPE] [--db-location DB_LOCATION] [--db
           -name DB_NAME] [--db-type DB_TYPE] [--limit LIMIT]

Usage: ./army-ant.py search QUERY [OFFSET] [LIMIT] [INDEX_LOCATION] [INDEX_TYPE] [DB_LOCATION]
    [DB_NAME] [DB_TYPE]
       ./army-ant.py search --query QUERY [--offset OFFSET] [--limit LIMIT] [--index-location
           INDEX_LOCATION] [--index-type INDEX_TYPE] [--db-location DB_LOCATION] [--db-name
           DB_NAME] [--db-type DB_TYPE]

Usage: ./army-ant.py fetch-wikipedia-images DB_NAME [DB_LOCATION] [DB_TYPE]
       ./army-ant.py fetch-wikipedia-images --db-name DB_NAME [--db-location DB_LOCATION] [--db
           -type DB_TYPE]

Usage: ./army-ant.py server
```

implementations of graph-of-word and graph-of-entity. This way, the index location is dependent only on the index implementation. Finally, a `--limit` argument can be used to only read or retrieve the first n documents from a dataset. In the following paragraphs, we will present further details on the inner workings of each module.

Reader

The `army_ant.reader` package contains two main classes: `Reader` and `Document`. The `Reader` class must be extended for implementing a particular dataset iterator that returns `Document` instances. It can also be used to instantiate readers, given a `source_reader` argument to the `factory()` method, which must also be extended with a new `elif` condition to instantiate any new implemented reader. Implementing a new `Reader` consists of overriding the `__next__()` method that must either return the next `Document` from the dataset or raise a `StopIteration` exception indicating that it has finished iterating. This first version of Army ANT already implements two readers: `WikipediaDataReader`, for the Wikipedia Relation Extraction Data v1.0[4] (Culotta, McCallum, and Betz [2006]) and `INEXReader`, for the INEX 2009 Collection[5] (Schenkel, Suchanek, and Kasneci [2007]). Both readers share a `Wikipedia-Entity` class that will, later on, be made available generally to any other reader that indexes a knowledge block, at which point it will simply be renamed to `Entity` in resemblance to `Document`.

Index

The `army_ant.index` package contains two main classes: `Index` and `ServiceIndex`. The `Index` class, which also acts as a factory to instantiate any supported index, must be extended for implementing a particular engine by overriding the `index()` and `search()` methods. Alternatively, a `ServiceIndex` class can be extended instead, in order to obtain an `index_host` and an `index_port` property from the `index_location` (e.g., "localhost:8182"). The `index()` method can, optionally, yield instances of `Document` that will then be passed to a `Database` instance to store document metadata. Each implemented `Index` or engine must use Python's new `async` API, since this will then be served via `aiohttp` in the `army_ant.server` package. This first version of Army ANT already implements two engines: `GraphOfWord` and `GraphOfEntity`. These retrieval models are detailed in Chapter 3, however there is a relevant technical detail about either model that we will share here. While `GraphOfWord` yields a `Document` instance directly from the reader, as it focuses on indexing text documents, `GraphOfEntity`

[4]http://cs.iit.edu/~culotta/data/wikipedia.html
[5]http://www.mpi-inf.mpg.de/departments/databases-and-information-systems/software/inex/

Listing 2.2: Army ANT `MongoDatabase` example of a JSON document in the `documents` collection.

```
{
  "_id" : ObjectId("591dad4c02f3ddaa2d990d1b"),
  "doc_id" : "7577000",
  "metadata" : {
      "url" : "http://en.wikipedia.org/?curid=7577000",
      "name" : "Johnny Burke"
    }
}
```

instead yields a `Document` instance for the subject entity in the list of triples, consisting of the same `doc_id` as the original document and providing metadata for the `url` and `name` of the entity. We might say that the retrieval unit for the graph-of-word is a text document, while the retrieval unit for the graph-of-entity is an entity. In this case, however, both are represented by a URL associated with the respective Wikipedia article.

Database

The `army_ant.database` package contains a single main class: `Database`. The `Database` class, which also acts as a factory to instantiate any supported metadata databases, must be extended for implementing a facade to access the storage layer by overriding the `store()` and `retrieve()` methods, which take an `Index` and a results list from an `Index.search()` method, respectively. Storing and retrieving batches of documents, instead of individual documents, gives a chance to the storage layer for improving performance with transactions and batch operations. This first version of Army ANT already implements a database facade for `MongoDatabase`. It stores metadata in a `documents` collection, following the schema illustrated in Listing 2.2.

Evaluation

The `army_ant.evaluation` package contains three main classes: `EvaluationTaskManager`, `EvaluationTask` and `Evaluator`. The `EvaluationTaskManager` class is called by the `army_ant.server` module to add a task to a batch via `add_task()`, to queue all added tasks via `queue()`, to process the next task via `process()` or to list queued tasks via `get_tasks()`. It also manages other lower level activities like removing unreferenced files or reseting the status of running tasks in the event of a server interruption. The `EvaluationTask` is a data class to store the following fields associated with an evaluation task: `topics_filename` (original filename when uploaded), `topics_path` (internal path of the file, pointing to a temporary file in a spool directory), `topics_md5`, `assessments_filename`, `assessments_path`, `assessments_md5`, `index_location`, `index_type` (currently “gow” and “goe” are supported), `eval_format` (currently “inex” is supported, with plans to also implement a “trec” evaluator), `status` (an `IntEnum` implemented through `EvaluationTaskStatus` with values for `WAITING`, `RUNNING` and `DONE`), `time` (the timestamp for the queue time), `results` (only for `DONE` tasks) and `_id` (only for tasks loaded from the database). Finally, the `Evaluator` class, which also acts as a factory to instantiate any supported evaluator, must be extended for implementing a particular evaluation format by overriding the `run()` method, which must generate an output CSV on the results directory, with three columns: `rank` (positive integer), `doc_id` (string or number) and `relevant` (“True” or “False” string values). It can also generate a series of CSV files in the assessments directory, with intermediate data required by each computed evaluation metric. A `results` property can be set during any stage of the run, containing a map of $metric \mapsto value$ to be displayed in the web interface. This first version of Army ANT already implements the `INEXEvaluator` for the topic and assessment files of the INEX 2009 Collection. In particular, this iterates over the `title` child element (the query) of each `topic` element, creating a results file named with the `id` attribute of the `topic` element. Results are then assessed using a binary grade (relevant/not relevant), based on the relevance judgments and, in particular for INEX, on whether the number of relevant characters was zero (not relevant) or larger then zero (relevant). Based on this information, we then calculate MAP and NDCG@10, which we store in the database; any intermediate files are stored in the evaluation output directory and can then be downloaded as a ZIP file, along with a CSV with the values for the two evaluation metrics.

Server

The `army_ant.server` package implements an `aiohttp`[6] server. HTTP methods are directly defined within the `__init__.py` of the module, using the `templates` directory to store Jinja2[7] HTML templates and the `static` directory to store `css`, `img` or `js` files. CSS and JavaScript web packages are managed by NPM[8] through a `packages.json` file on the root directory of the project. The server module implements the following HTTP requests: `GET /` for the home page, `GET /search` for the search and learn mode interface, `GET /evaluation` to manage evaluation tasks, `POST /evaluation` to launch a new evaluation task, `GET /evaluation/results` to download ZIP files containing the output of an evaluation task, `GET /about` with information about Army ANT and `GET /static` to serve static files. The server also runs the `EvaluationTaskManager.manage()` method, which periodically launches `WAIT`ing tasks.

Util

The `army_ant.util` package simply implements functions that can be reused throughout the application and currently includes, for this first version of Army ANT: `html_to_text()` to strip HTML tags from an HTML fragment string, `load_gremlin_script()` to load a Gremlin script as a string, `md5()` to compute the MD5 of a file given its filename, `get_first()` to get the first element of a list or `None` if empty, and `zipdir` to create a ZIP archive of a given directory.

Extras

The `army_ant.extras` package provides additional functions that are frequently called from the command line interface and that are particular to a given dataset or engine implementation. In this first version of Army ANT, we provide the procedure `fetch_wikipedia_images()` to complement metadata for Wikipedia documents with the first image in the infobox of the corresponding web page. This enables us to display a thumbnail next to a Wikipedia result within the web interface.

User's Manual

In this section, we will provide an overview of the requirements for a normal operation of Army ANT, including any dependencies for the implemented classes. We will then provide examples for the command line interface, in particular listing supported values for the arguments of implemented readers, engines and extras. Finally, we will describe the web interface and how it can be used to learn about, explore and evaluate implemented engines.

Requirements and Installation

In order to run Army ANT, you first require an installation of Python 3.6.x. Since the latest version of Python is not widely available in the software repositories for the main Linux distributions, we instead managed the Python version through pyenv[9]. As of the writing of this report, pyenv should be installed and configured as follows:

```
git clone https://github.com/pyenv/pyenv.git ~/.pyenv
echo 'export PYENV_ROOT="$HOME/.pyenv"' >> ~/.bashrc
```

[6] `http://aiohttp.readthedocs.io/en/stable/`
[7] `http://jinja.pocoo.org/`
[8] `https://www.npmjs.com/`
[9] `https://github.com/pyenv/pyenv#installation`

```
echo 'export PATH="$PYENV_ROOT/bin:$PATH"' >> ~/.bashrc
echo 'eval "$(pyenv init -)"' >> ~/.bashrc
```

After installing pyenv and restarting the terminal emulator to load the `~/.bashrc` script (or simply by running `source ~/.bashrc`, Python 3.6.0 will automatically be used within the Army ANT directory, as configured in the `.python-version` file. At this point, you should run the following command, in the root of the project, to install Python dependencies:

```
pip install -r requirements.txt
```

For the `MongoDatabase`, you will also need a MongoDB instance running. In a Debian based distribution, you can simply run the following command to do this:

```
sudo apt-get install mongodb-server
sudo service mongodb start
```

For the provided `GraphOfWord` and `GraphOfEntity` implementations, you will also need Java 8 and Apache TinkerPop Gremlin Server. Java 8 can be installed through the following command:

```
sudo apt-get install openjdk-8-jdk
```

You can install Apache TinkerPop Gremlin server by visiting the website[10], downloading the binary for the 3.2.4 version of the Gremlin Server and uncompressing it in `/opt/apache-tinkerpop-gremlin-server-3.2.4`. The Neo4j Plugin must then be installed, otherwise the server will not provide any persistence and will not work with the provided configurations:

```
cd /opt/apache-tinkerpop-gremlin-server-3.2.4
bin/gremlin-server.sh -i org.apache.tinkerpop neo4j-gremlin 3.2.4
```

In the event that the installation fails, try to create the Groovy configuration file `~/.groovy/groovyConfig.xml`, with the following contents, in order to reconfigure the download location for Grapes (the Groovy dependency management software) and retry the previous command:

```
<ivysettings>
  <settings defaultResolver="downloadGrapes"/>
  <resolvers>
    <chain name="downloadGrapes">
      <filesystem name="cachedGrapes">
        <ivy pattern="${user.home}/.groovy/grapes/[organisation]/[module]/ivy-[revision].xml"/>
        <artifact pattern="${user.home}/.groovy/grapes/[organisation]/[module]/[type]s/[artifact
            ]-[revision].[ext]"/>
      </filesystem>
      <ibiblio name="codehaus" root="http://repository.codehaus.org/" m2compatible="true"/>
      <ibiblio name="central" root="http://central.maven.org/maven2/" m2compatible="true"/>
      <ibiblio name="jitpack" root="https://jitpack.io" m2compatible="true"/>
      <ibiblio name="java.net2" root="http://download.java.net/maven/2/" m2compatible="true"/>
    </chain>
  </resolvers>
</ivysettings>
```

After installing the Neo4j Plugin, we suggest you create symbolic links to the provided configuration files in `/opt/apache-tinkerpop-gremlin-server-3.2.4/conf`:

[10] https://tinkerpop.apache.org/

```
cd /opt/apache-tinkerpop-gremlin-server-3.2.4/conf
ln -s <Army ANT directory>/config/gremlin-server-neo4j-graph-of-word-wikipedia.yaml
ln -s <Army ANT directory>/config/gremlin-server-neo4j-graph-of-entity-wikipedia.yaml
ln -s <Army ANT directory>/config/gremlin-server-neo4j-graph-of-word-inex.yaml
ln -s <Army ANT directory>/config/gremlin-server-neo4j-graph-of-entity-inex.yaml
ln -s <Army ANT directory>/config/neo4j-graph-of-word-wikipedia.properties
ln -s <Army ANT directory>/config/neo4j-graph-of-entity-wikipedia.properties
ln -s <Army ANT directory>/config/neo4j-graph-of-word-inex.properties
ln -s <Army ANT directory>/config/neo4j-graph-of-entity-inex.properties
```

Finally, create the directory structure for Army ANT on `/opt/army-ant`:

```
mkdir /opt/army-ant
mkdir /opt/army-ant/{data,eval}
mkdir /opt/army-ant/eval/{spool,results,assessment}
```

You can now try to run for instance Army ANT server to ensure everything went right:

```
cd <Army ANT directory>
./army-ant.py server
```

In the following sections, we will present further details on how to use the command line interface to index one of the supported collections using one of the implemented engines, as well as to use the web interface to issues queries and explore results to the indexed collections and to evaluate different engines over the INEX 2009 Collection.

Command Line Interface

In this section, we will illustrate the four commands supported by `army-ant.py`, presenting a complete example for each of them and a list of the supported values for each argument.

Index

The first step in a workflow, assuming a reader and an engine has already been implemented, is to index a supported dataset. Next, we exemplify the indexing action based on the INEX 2009 Collection, the graph-of-word index type and a MongoDB instance to store metadata:

```
cd <Army ANT directory>
./army-ant.py index --source-path "INEX 2009/dataset/pages25.tar.bz2" --source-reader "inex" --
    index-location "localhost:8184" --index-type "gow" --db-location "localhost:27017" --db-
    name "graph_of_word_inex" --db-type "mongo"
```

Valid `--source-reader` values include `wikipedia_data` for Wikipedia Relation Extraction Data v1.0 and `inex` for INEX 2009 Collection. Valid `--index-type` values include `gow` for the graph-of-word engine and `goe` for the graph-of-entity engine. Valid `--db-type` values only include `mongo`. Any of these adapted for different dataset sources, retrieval models and storage layers, respectively, by extending the base classes as described previously in Section 2.1.

Search

While the command line is not the preferred search interface (we suggest you use the web interface instead, for obvious reasons), Army ANT still provides some facilities to test the engine via the command line. Next, we present an example of a search query to the graph-of-word engine for the INEX 2009 Collection:

```
cd <Army ANT directory>
./army-ant.py search --query "york" --offset 0 --limit 3 --index-location "localhost:8184" --
    index-type "gow" --db-location "localhost:27017" --db-name "graph_of_word_inex" --db-type "
    mongo"
```

This will result in the following output, with the rank, score, document identifier and metadata for each returned item:

```
===> 1 75.08 193001
             url: http://en.wikipedia.org/?curid=193001
            name: Louis Brandeis

===> 2 56.76 15334002
             url: http://en.wikipedia.org/?curid=15334002
            name: New York State Route 323

===> 3 50.02 19346003
             url: http://en.wikipedia.org/?curid=19346003
            name: Bond Clothing Stores
```

The same set of valid values described for the index command can be used with the search command, specifically for `--index-type` and `--db-type`.

Server

Listing 2.3: Army ANT server configuration file for a graph-of-word and a graph-of-entity index per dataset: Wikipedia Relation Extraction Data v1.0 and INEX 2009 Collection.

```
[DEFAULT]
db_location=localhost
db_type=mongo
eval_location=/opt/army-ant/eval

[gow-wikipedia_data]
name=Wikipedia Data - Graph of Word
index_type=gow
index_location=localhost:8182
db_name=graph_of_word_wikipedia

[goe-wikipedia_data]
name=Wikipedia Data - Graph of Entity
index_type=goe
index_location=localhost:8183
db_name=graph_of_entity_wikipedia

[gow-inex]
name=INEX - Graph of Word
index_type=gow
index_location=localhost:8184
db_name=graph_of_word_inex

[goe-inex]
name=INEX - Graph of Entity
index_type=goe
index_location=localhost:8185
db_name=graph_of_entity_inex
```

In order to run the server and access the learning mode and evaluation modules, you must first configure a set of valid engines and other parameters through the `server.cfg` file. Listing 2.3 presents an example of the configuration file used during this experience. The properties defined in `DEFAULT` section are shared by the remaining sections, unless redefined in that section. In particular, we use the localhost instance of MongoDB on the default port (unspecified). We also use the `/opt/army-ant/eval` directory as the base directory for evaluation task input and output. Besides `DEFAULT`, each of the other sections contains the configuration for an engine, with the name of section defined in the format `<engine>-<dataset>` and directly used as the value of the selection dropdown in the web interface. The `name` of the engine is used to display each option in the dropdown to the user. The `index_type` property determines the engine to use and the `index_location` property defines the location of the index of the given type, for the particular dataset it refers to. Finally, `db_name` simply points to the MongoDB database that stores metadata about the documents. While, in this example, we use different instances for all engines, we could have used the same database per dataset, since both indexes would reference the same document collection. The server, which runs in port 8080, can then be run using the following command:

```
./army-ant.py server
```

Extras

As we have previously described in Section 2.1, we have only implemented one extra feature that can be called through the command line interface with the `fetch-wikipedia-images` command, as shown below:

```
./army-ant.py fetch-wikipedia-images --db-name "graph_of_word_inex" --db-location "localhost
    :27017" --db-type "mongo"
```

This will read a `url` metadata attribute and, if it matches a Wikipedia URL, visit the corresponding web page and extract the first image from the infobox table, if any exists. The URL for the image is then stored in the `img_url` metadata attribute and will be used, when available, to display an image next to each search result.

Web Interface

In this section, we describe the web interface and how it can be used to learn about, explore and evaluate implemented retrieval models (engines). This can only be done after implementing the required reader, index and evaluator classes, and indexing any relevant collections. First, we describe the basic search interface, not unlike a common search engine, with the exception of the ability to select a retrieval model to search with and an indexed dataset to search over. Then, we describe the learn mode interface, where information for the first 30 results is displayed, with the goal of explaining how the scores for each document were calculated. This also includes bibliographic references to scientific publications describing the retrieval model (document representation and weighting scheme) and the indexed dataset. Finally, we explain how the evaluation interface works, presenting further details on accepted formats and generated output.

Search

When you first access Army ANT server, for example on `http://localhost:8080`, you will be taken to a Home page, suggesting you visit the About page for further information. From the Home page, you can also access the search and learn mode interface via the Search anchor, or the evaluation interface via the Evaluation anchor. When you first press the Search anchor, you will be taken to an empty search page with a query input box, an engine selection dropdown and a learn mode toggle button. Let us say we wanted to search for `born new york`, using the graph-of-entity retrieval model over the Wikipedia Relation Extraction Data v1.0. The result of this action is illustrated in Figure 2.2. As we can see, we

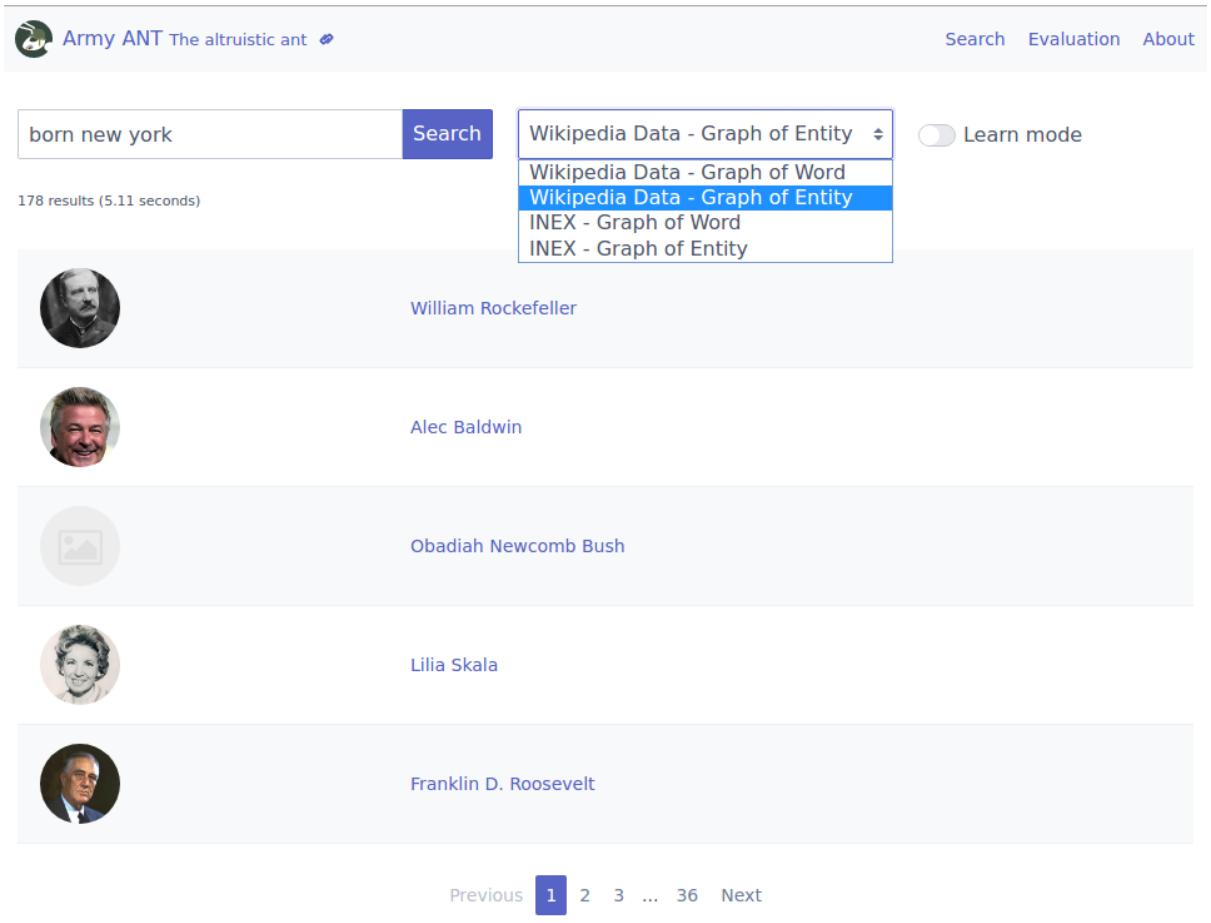

Army ANT The altruistic ant
Search Evaluation About
born new york
Search
Wikipedia Data - Graph of Entity
Learn mode
178 results (5.11 seconds)
Wikipedia Data - Graph of Word
Wikipedia Data - Graph of Entity
INEX - Graph of Word
INEX - Graph of Entity
William Rockefeller
Alec Baldwin
Obadiah Newcomb Bush
Lilia Skala
Franklin D. Roosevelt
Previous 1 2 3 ... 36 Next

Figure 2.2: Army ANT basic search and engine selection.

present a clean interface, paginating results and providing an image and link to each web document when possible. This provides a basic interface to explore indexed datasets with different retrieval models.

Learn Mode

The learn mode was created with two goals in mind. First, it can be used to help in the design of a retrieval model by explaining a document's score through its individual components. Secondly, it can be used to teach information retrieval, as an interactive documentation platform for retrieval models and document collections. It clearly shows how each particular weighting function works and provides further web and bibliographic references on the subject, as well as on the document collection, aiding with understanding the structure of indexed contents.

Parallel Coordinates Plot Figure 2.3 shows a visualization of individual score components — we chose the parallel coordinates system (Inselberg [1985]), since it allows for a condensed and useful view of multivariate data. This kind of plot acts as a decision support mechanism, in order to choose whether a component is redundant, has too low or too high of an impact, requiring scaling or normalization, and to build an intuition on the discriminative power of a score component.

Score As a complement to the parallel coordinates plot, we also provide descriptive content on the score components, fully describing the weighting function and providing a bibliographic citation and corresponding link to the publication that details that particular retrieval model, usually also covering the document representation model. Figure 2.4 illustrates such information for the graph-of-word retrieval model.

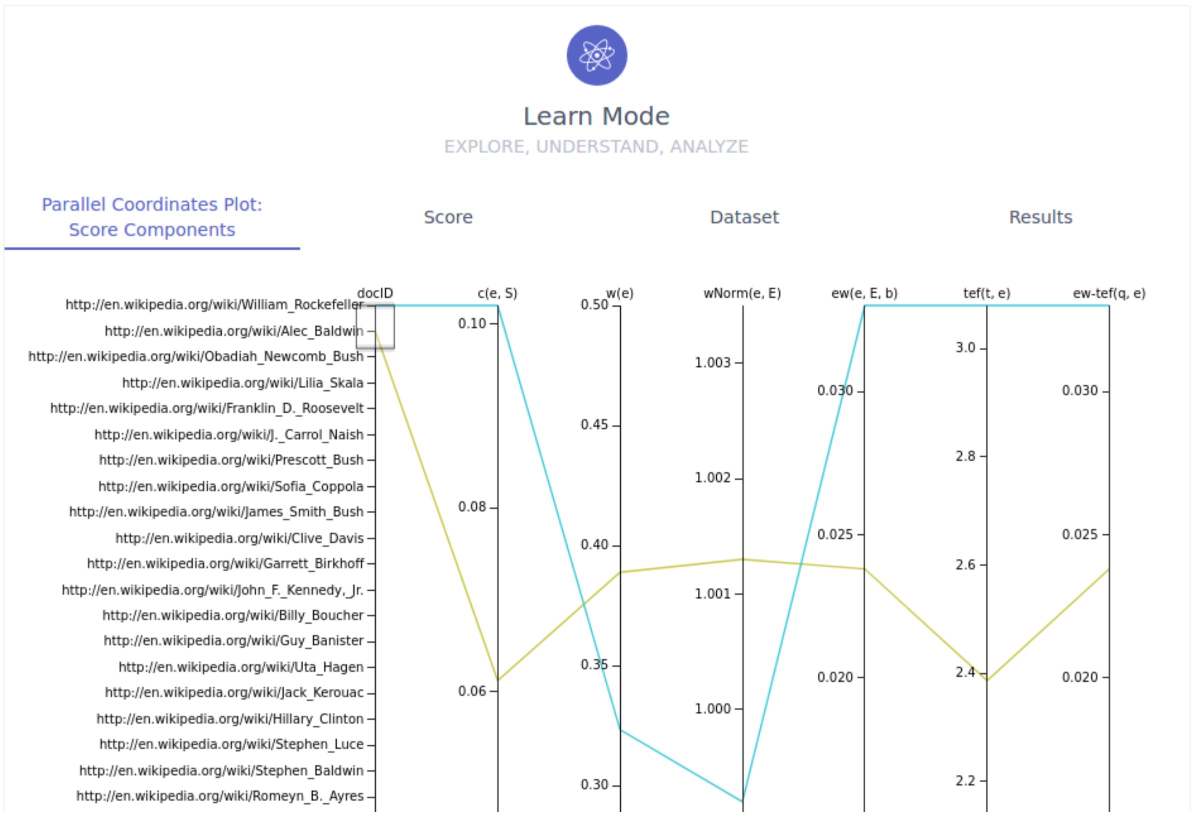

Learn Mode
EXPLORE, UNDERSTAND, ANALYZE
Parallel Coordinates Plot: Score Components
Score
Dataset
Results
docID
c(e, S)
w(e)
wNorm(e, E)
ew(e, E, b)
tef(t, e)
ew-tef(q, e)
http://en.wikipedia.org/wiki/William_Rockefeller
http://en.wikipedia.org/wiki/Alec_Baldwin
http://en.wikipedia.org/wiki/Obadiah_Newcomb_Bush
http://en.wikipedia.org/wiki/Lilia_Skala
http://en.wikipedia.org/wiki/Franklin_D._Roosevelt
http://en.wikipedia.org/wiki/J._Carrol_Naish
http://en.wikipedia.org/wiki/Prescott_Bush
http://en.wikipedia.org/wiki/Sofia_Coppola
http://en.wikipedia.org/wiki/James_Smith_Bush
http://en.wikipedia.org/wiki/Clive_Davis
http://en.wikipedia.org/wiki/Garrett_Birkhoff
http://en.wikipedia.org/wiki/John_F._Kennedy,_Jr.
http://en.wikipedia.org/wiki/Billy_Boucher
http://en.wikipedia.org/wiki/Guy_Banister
http://en.wikipedia.org/wiki/Uta_Hagen
http://en.wikipedia.org/wiki/Jack_Kerouac
http://en.wikipedia.org/wiki/Hillary_Clinton
http://en.wikipedia.org/wiki/Stephen_Luce
http://en.wikipedia.org/wiki/Stephen_Baldwin
http://en.wikipedia.org/wiki/Romeyn_B._Ayres
0.10
0.08
0.06
0.50
0.45
0.40
0.35
0.30
1.003
1.002
1.001
1.000
0.030
0.025
0.020
3.0
2.8
2.6
2.4
2.2
0.030
0.025
0.020

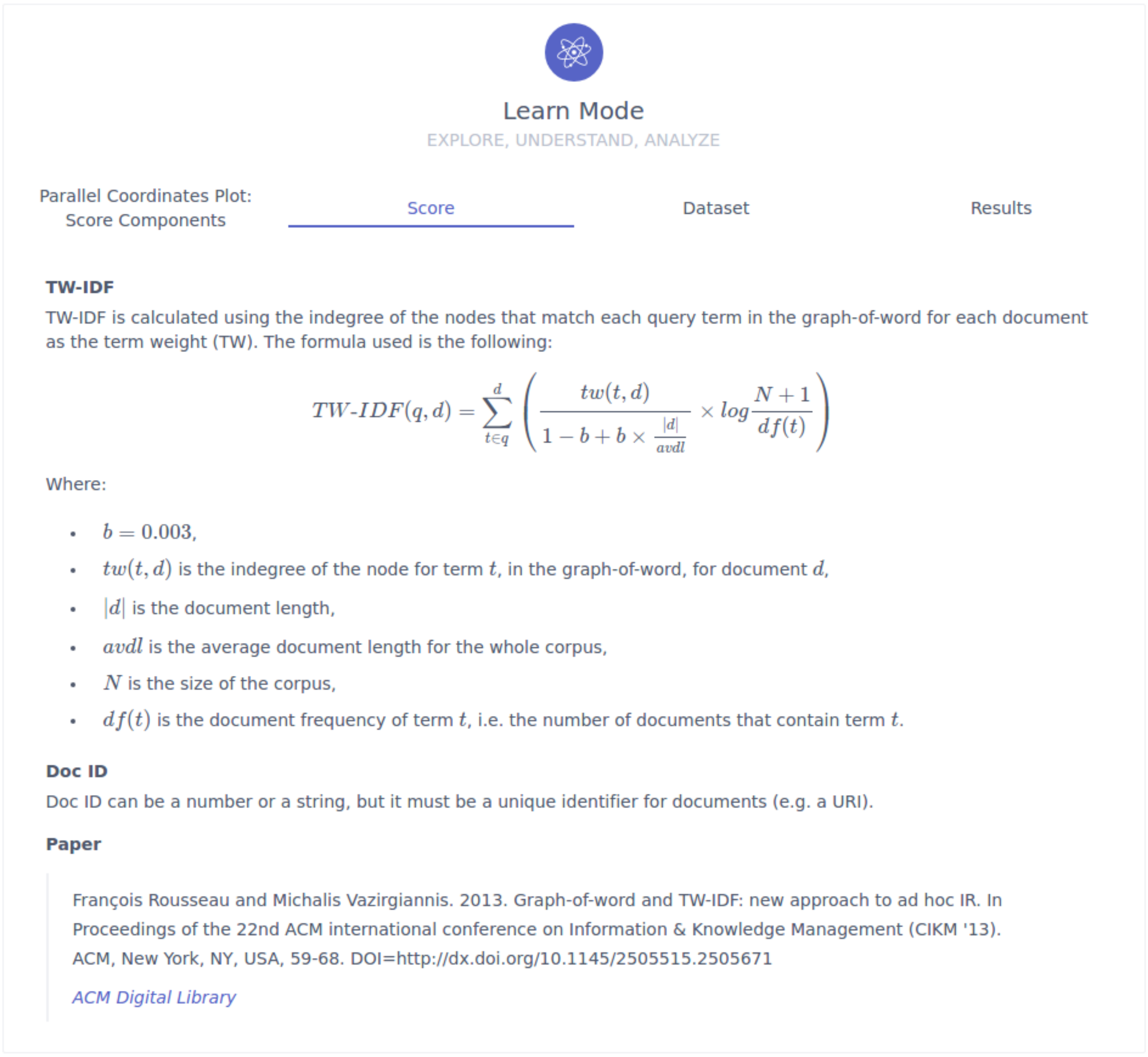
Figure 2.3: Army ANT learn mode parallel coordinates plot for score components per document.

Dataset Similarly, we also provide descriptive content on the indexed dataset, including a link to its web page, usually providing a download location, when available. We then present a short description of the collection, along with an example of the structure and content of a particular document. When available, we provide a bibliographic citation and corresponding link to the main publication associated with the dataset. This usually corresponds to a data collection and characterization paper, but sometimes it only describes a particularly relevant usage of the dataset.

Results Finally, as illustrated in Figure 2.7, we present the ranked list of results, displaying the rank position, the computed score and the document identifier, stripped of any metadata. This is useful in clarifying the difference between indexing and storage. Indexing solutions like Apache Solr have lately been used as databases, since documents are analyzed and indexed but also stored within this software. This has made the two processes harder to distinguish for a new student of information retrieval and, therefore, showing the results without any metadata is a useful tool to clarify this. While solutions like Apache Lucene/Solr also implement a kind of document database storage layer, the index is not, per se, a database! In fact, in Army ANT, we separate the two layers — nevertheless, it is useful to mention that this logical segmentation of indexing and storage might result in a slight loss of performance unless, of course, there is no interest in retrieving document metadata. This is because, at a low level, the index might directly store the offset of the metadata to retrieve for a document, while separating this would require a database index or a sequential search to reach the same metadata. For educative purposes, however, we believe this is the right choice, to separate the two layers.

Evaluation

Finally, we describe the evaluation interface, which requires the prior indexing of a dataset with relevance judgments for a set of search topics. In particular, the first version of Army ANT supports the `inex` evaluation format. This means that, in order to launch an evaluation task, we must first select the topics and assessments files, picking the INEX evaluator format and one of the available INEX engines



Learn Mode
EXPLORE, UNDERSTAND, ANALYZE
Parallel Coordinates Plot: Score Components
Score
Dataset
Results

TW-IDF

TW-IDF is calculated using the indegree of the nodes that match each query term in the graph-of-word for each document as the term weight (TW). The formula used is the following:

$$TW\text{-}IDF(q,d) = \sum_{t \in q}^{d} \left(\frac{tw(t,d)}{1 - b + b \times \frac{|d|}{avdl}} \times log \frac{N+1}{df(t)} \right)$$

Where:

- $b = 0.003$,
- $tw(t,d)$ is the indegree of the node for term t, in the graph-of-word, for document d,
- $|d|$ is the document length,
- $avdl$ is the average document length for the whole corpus,
- N is the size of the corpus,
- $df(t)$ is the document frequency of term t, i.e. the number of documents that contain term t.

Doc ID

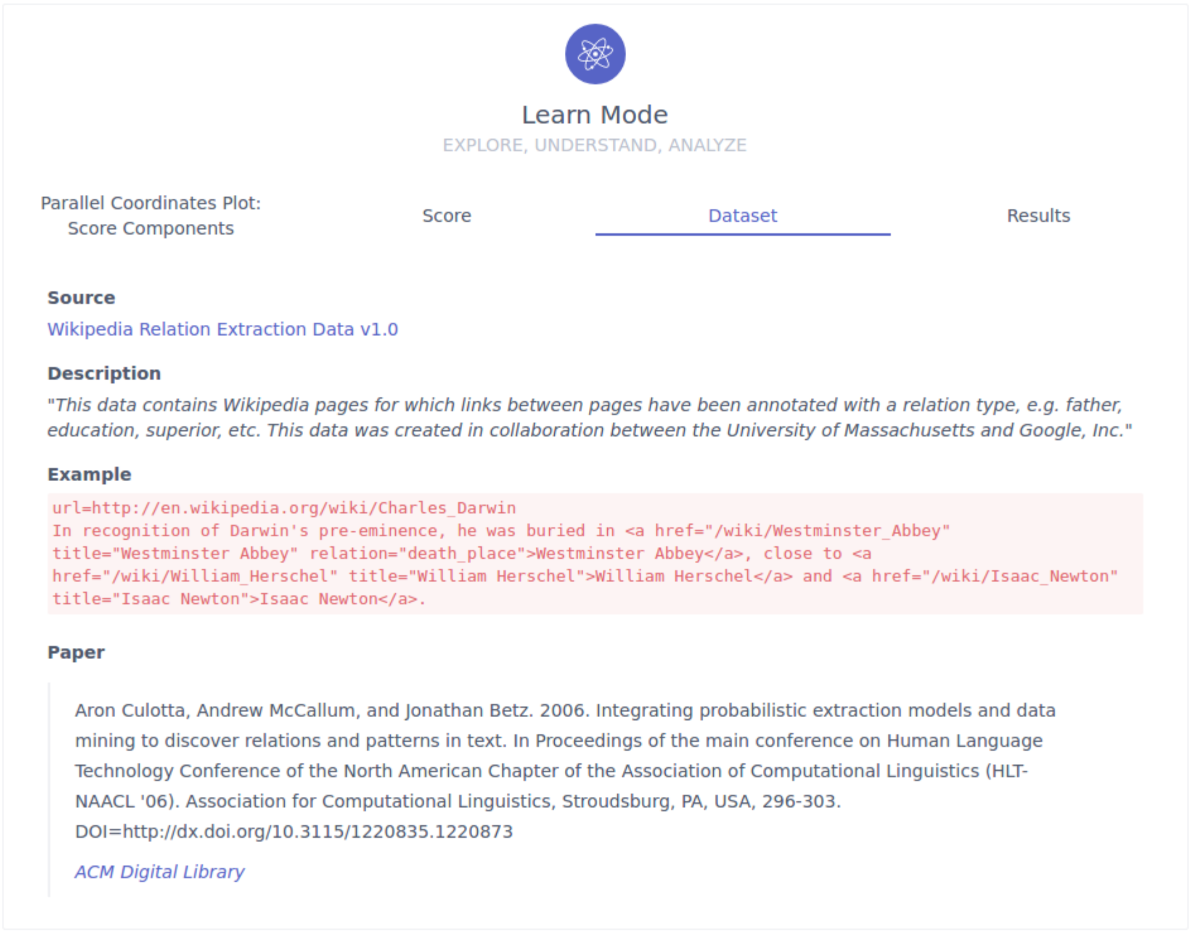
Doc ID can be a number or a string, but it must be a unique identifier for documents (e.g. a URI).

Paper

> François Rousseau and Michalis Vazirgiannis. 2013. Graph-of-word and TW-IDF: new approach to ad hoc IR. In Proceedings of the 22nd ACM international conference on Information & Knowledge Management (CIKM '13). ACM, New York, NY, USA, 59-68. DOI=http://dx.doi.org/10.1145/2505515.2505671
>
> *ACM Digital Library*

Figure 2.4: Army ANT learn mode descriptive explanation of the ranking function.

to evaluate. In the current version, only one evaluator can be active per topics file, assessments file and index, but this will changed in the future. The ability to delete evaluation tasks will also be added. An evaluation task has three possible statuses: when it's queued, it enters the queue in the `WAIT` status and, when the `EventTaskManager.process()` method is periodically called, the next available task will be started (only one task can currently run at a time). At this point, the task will change its status to `RUNNING` and, when it finishes, it will change its status to `DONE`. Finished tasks, enable an interface button with a plus/minus to toggle the results of the evaluation, in the form of a dictionary and usually displaying the values for MAP and NDGC@10. Additionally, the user can download a ZIP file containing a CSV with these metrics, as well as intermediate files used to compute the metrics, by pressing the download link in a `DONE` status.


Learn Mode
EXPLORE, UNDERSTAND, ANALYZE
Parallel Coordinates Plot: Score Components
Score
Dataset
Results

Source

Wikipedia Relation Extraction Data v1.0

Description

"This data contains Wikipedia pages for which links between pages have been annotated with a relation type, e.g. father, education, superior, etc. This data was created in collaboration between the University of Massachusetts and Google, Inc."

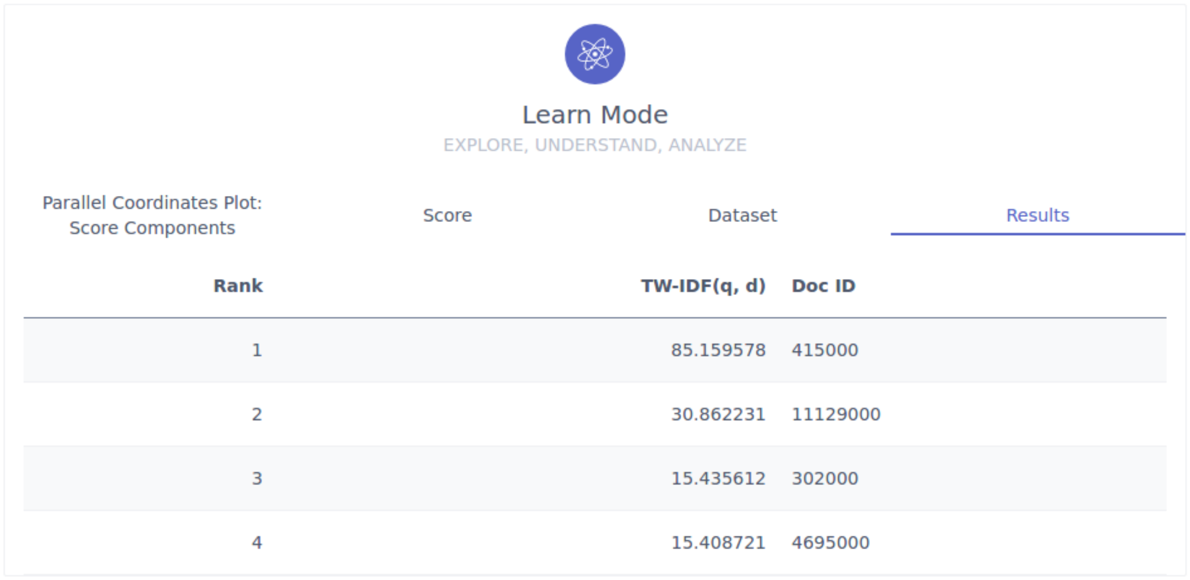
Example

```
url=http://en.wikipedia.org/wiki/Charles_Darwin
In recognition of Darwin's pre-eminence, he was buried in <a href="/wiki/Westminster_Abbey"
title="Westminster Abbey" relation="death_place">Westminster Abbey</a>, close to <a
href="/wiki/William_Herschel" title="William Herschel">William Herschel</a> and <a href="/wiki/Isaac_Newton"
title="Isaac Newton">Isaac Newton</a>.
```

Paper

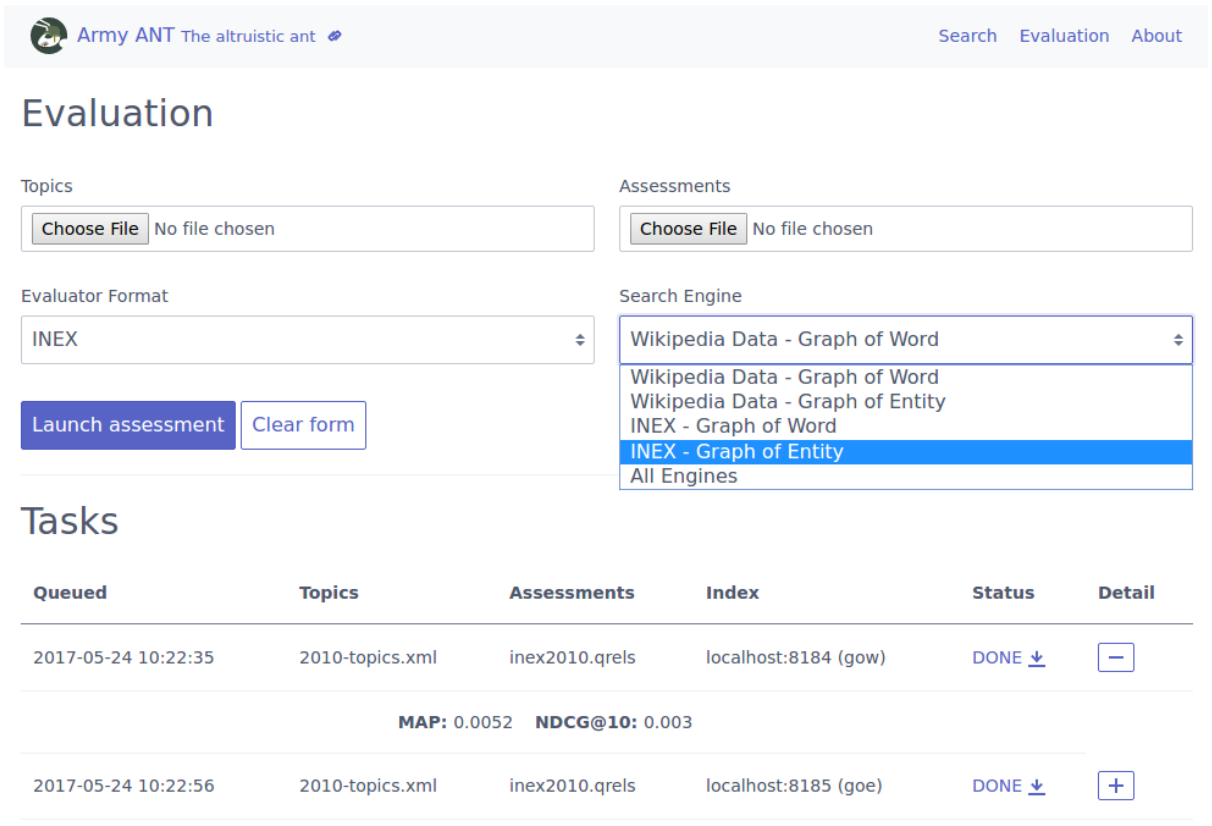
> Aron Culotta, Andrew McCallum, and Jonathan Betz. 2006. Integrating probabilistic extraction models and data mining to discover relations and patterns in text. In Proceedings of the main conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics (HLT-NAACL '06). Association for Computational Linguistics, Stroudsburg, PA, USA, 296-303. DOI=http://dx.doi.org/10.3115/1220835.1220873
>
> *ACM Digital Library*

Figure 2.5: Army ANT learn mode descriptive explanation of the indexed dataset(s).


Learn Mode
EXPLORE, UNDERSTAND, ANALYZE
Parallel Coordinates Plot: Score Components
Score
Dataset
Results

Rank	TW-IDF(q, d)	Doc ID
1	85.159578	415000
2	30.862231	11129000
3	15.435612	302000
4	15.408721	4695000

Figure 2.6: Army ANT learn mode detailed view of the top-30 results, without metadata.



Evaluation

Topics: Choose File No file chosen

Assessments: Choose File No file chosen

Evaluator Format: INEX

Search Engine: Wikipedia Data - Graph of Word

- Wikipedia Data - Graph of Word
- Wikipedia Data - Graph of Entity
- INEX - Graph of Word
- INEX - Graph of Entity
- All Engines

Tasks

Queued	Topics	Assessments	Index	Status	Detail
2017-05-24 10:22:35	2010-topics.xml	inex2010.qrels	localhost:8184 (gow)	DONE	−
		MAP: 0.0052 **NDCG@10:** 0.003			
2017-05-24 10:22:56	2010-topics.xml	inex2010.qrels	localhost:8185 (goe)	DONE	+

Figure 2.7: Army ANT learn mode evaluation task manager.

Chapter 3

Graph-Based Models

As a net is made up of a series of ties, so everything in this world is connected by a series of ties. If anyone thinks that the mesh of a net is an independent, isolated thing, he is mistaken. It is called a net because it is made up of a series of an interconnected meshes, and each mesh has its place and responsibility in relation to other meshes.

— Gautama Buddha

In this chapter, we describe the two graph-based document representation and retrieval models implemented in Army ANT. First, we cover the graph-of-word model, proposed by Rousseau and Vazirgiannis (2013), along with our implementation of the model. We used a common graph structure for all documents, built for the sake of generalization within our explorative approach. This was stored on a graph database and queried through the Gremlin[1] domain-specific language for graphs. We also propose a novel graph-of-entity model, where we extend the notion of document beyond textual content, associating text with a set of statements that portray knowledge related to the text. This was an attempt to structure and more clearly define the task of entity-oriented search, looking at the document as a richer source of information. The graph-of-entity is therefore an early attempt at integrating different information sources, seamlessly and in a unified model, where text, entities and their relations are included in a common graph structure. While we certainly sacrifice efficiency, by using a graph database instead of an inverted file to index the documents, our goal is to be able to explore graph-based approaches, with a focus on benchmarking effectiveness, in order to determine whether it is worth it to further invest on developing graph-based index data structures.

In the following sections, we will begin by describing the two datasets we used to build and evaluate the two models. We will introduce the graph-of-word, describing graph-based document representation and comparing the original implementation based on an inverted file with our implementation based on a graph database. We then describe the process of doing document retrieval on the graph database, as well as the graph-of-word ranking function and the information retrieval theory behind it. We follow this with a description of the graph-of-entity model, as a proposal to extend the graph-of-word with a knowledge block of entities and their relations in connection to the text. We then show how we can take advantage of graph querying to unify the information extraction task of query entity linking with the information retrieval task of entity ranking. We close the chapter with the evaluation of the two retrieval models, using MAP and NDCG@10 for the topics and relevance judgments of the INEX 2009 Collection.

Datasets for Combined Data

In order to experiment with entity-oriented search, we first had to obtain a dataset. We had two options to do this. We could find a collection of textual documents and then apply information extraction

[1] `http://tinkerpop.apache.org/docs/current/reference/#_on_gremlin_language_variants`

Listing 3.1: Example of Wikipedia Relation Extraction Data v1.0.

```
url=http://en.wikipedia.org/wiki/Charles_Darwin
In recognition of Darwin's pre-eminence, he was buried in <a href="/wiki/Westminster_Abbey"
title="Westminster Abbey" relation="death_place">Westminster Abbey</a>, close to <a href="/wiki
/William_Herschel" title="William Herschel">William Herschel</a> and <a href="/wiki/
Isaac_Newton" title="Isaac Newton">Isaac Newton</a>.

url=http://en.wikipedia.org/wiki/John_Quincy_Adams
Adams's most important contributions to American history came before and after his relatively
ineffective term as President. Before becoming President, he was the most experienced diplomat
in the United States. While serving as <a href="/wiki/United_States_Secretary_of_State" title="
United States Secretary of State" relation="job_title">Secretary of State</a> under President <
a href="/wiki/James_Monroe" title="James Monroe" relation="superior">James Monroe</a>, Adams
negotiated the <a href="/wiki/Adams-On%C3%ADs_Treaty" title="Adams-Onís Treaty">Adams-Onís
Treaty</a> with <a href="/wiki/Spain" title="Spain">Spain</a> and devised the <a href="/wiki/
Monroe_Doctrine" title="Monroe Doctrine">Monroe Doctrine</a>, both of which were of long
lasting importance. For these activities he has been called "the most influential American
grand strategist of the nineteenth century" and "perhaps the greatest secretary of state in
American history."<span class="reference"><sup id="fn_1_back"><a href="#fn_1" title="">1</a></
sup></span>
```

techniques, like named entity recognition and relation extraction, to build the associated knowledge base. Or we could find a dataset of combined data that already provides knowledge (i.e., entities and relations) along with the textual content. In order to focus on the construction of the workbench and the study and implementation of the retrieval models, we chose the latter. We found a small dataset — Wikipedia Relation Extraction Data v1.0 — that fit the criteria and was ideal for the development stage and for continuous experimentation. Lacking topics and relevance judgments, we were required to find another dataset that provided this information, in order to evaluate the retrieval models. Another alternative would be to test the system with real users, but, given the time constraints, this was not an option. Furthermore, we used an already established dataset — INEX 2009 Collection —, which has been used to evaluate several information retrieval tasks, in competition tracks similar to TREC. Using this dataset also helped position the two tested models in regards to the state of the art. Next, we will describe the two datasets in more detail, identifying the parts that were indexed as text and the parts that were indexed as knowledge.

Wikipedia Relation Extraction Data v1.0

Wikipedia Relation Extraction Data v1.0[2] is freely provided by Aron Culotta, who leads TAPI Lab (Text Analysis in the Public Interest Laboratory). This dataset was created from a collaboration between the University of Massachusetts and Google, Inc. It contains two text files, `wikipedia.train` and `wikipedia.test`, with several passages (paragraphs) from Wikipedia pages. Anchors provided within each passage are annotated with the relation between the entities described by the current Wikipedia page and the target Wikipedia pages. Overall, it covers 441 Wikipedia pages (257 in the training set and 184 in the test set), 1110 passages (777 in the training set and 333 in the test set), 4681 relations (3332 in the training set and 1349 in the test set) and 53 relation types. Listing 3.1 shows the first two passages of the `wikipedia.train` file — for our experiments, we only indexed the training file. Each passage beings with its corresponding URL from Wikipedia and is followed by a paragraph containing an HTML fragment with `a` elements annotated with a special `relation` attribute.

There can be multiple passages per Wikipedia page. In Army ANT, we consider all the passages per Wikipedia page as a single document. We use the concatenated passages, stripped of HTML tags, as our text block, and associate each Wikipedia page with its implicit entity, as extracted from the URL ending, building a knowledge block based on the relations between such entities. During indexing, we

[2] `http://cs.iit.edu/~culotta/data/wikipedia.html`

Listing 3.2: Example of INEX 2009 Collection.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- generated by CLiX/Wiki2XML [MPI-Inf, MMCI@UdS] $LastChangedRevision: 92 $ on 17.04.2009
04:39:56[mciao0825] -->
<!DOCTYPE article SYSTEM "../article.dtd">
<article xmlns:xlink="http://www.w3.org/1999/xlink">
<header>
<title>Portal:Comedy/Did you know/61</title>
<id>18359000</id>
[...]
</header>
<bdy>

... that the <link xlink:type="simple" xlink:href="../616/44616.xml">
British radio sitcom</link> <b><it><link xlink:type="simple" xlink:href="../276/18249276.xml">
Safety Catch</link></it></b> is built around the moral dilemmas of a man who inadvertently
became an <link xlink:type="simple" xlink:href="../296/608296.xml">
arms dealer</link>?
[...]
</bdy>
</article>
```

annotate any retrievable instance of a document or entity with a `doc_id` equal to the URL, in order to distinguish between entities that are used for enriching the model versus entities that are to be retrieved. Ideally, we should not have to do this, but during our experiments we found that adding incomplete data (i.e., entities without associated text) would somehow imbalance the model and introduce noise that would decrease the quality of the results. While we did not measure this explicitly, it was clear from the results that, for the two studied models, the retrieval unit had to be properly identified. In the future, we would like to further explore this issue and generalize retrieval to any available information.

INEX 2009 Collection

While the focus was on measuring the effectiveness of the retrieval models, we had visible performance issues while indexing the INEX 2009 Collection. This is a larger dataset that comprises 50.7 GB of Wikipedia articles, in XML format, annotated with entity categories and internal links to other documents; it contains over 2.6 million documents, from October 8, 2008, with over 1.4 billion XML elements. Listing 3.2 shows the structure of document `18359000.xml` from `pages/000` of the `sample-25-000.tar.bz2` archive — each document has a `header` element, with metadata, and a `bdy` element, with the main content of the document. Out of all the available XML annotations, we have only taken into consideration `link` elements, in order to establish a relation from the current document to other documents in the collection. This was used as the knowledge block of the entity-oriented document, mapping Wikipedia pages to their implicit entity and considering only `related_to` relations to other entities. The textual block of the document consisted on the text of the `bdy` tag, stripped of XML tags. We indexed the textual block with the graph-of-text and both the textual and knowledge blocks with the graph-of-entity. The intent was to build a better graph-based model by also taking available knowledge into consideration. We were able to assess the quality of our proposed model based on the topics and relevance judgments provided along the INEX 2009 Collection.

Graph of Word

The graph-of-word model has been proposed by Rousseau and Vazirgiannis (2013) as a novel graph-based document representation, defying the term independence assumption of the bag-of-word approach. In particular, they used an unweighted directed graph (the graph-of-word), where nodes represent terms

Listing 3.3: Example document consisting of the first sentence of the "Semantic Search" Wikipedia page, with `DOCID=https://en.wikipedia.org/wiki/Semantic_search`.

```
<b>Semantic search</b> seeks to improve <a href="/wiki/Search_engine_technology" title="Search
    engine technology">search</a> accuracy by understanding the searcher's <a href="/wiki/
    Intention" title="Intention">intent</a> and the <a href="/wiki/Context_(language_use)"
    title="Context (language use)">contextual</a> meaning of terms as they appear in the
    searchable dataspace, whether on the <a href="/wiki/World_Wide_Web" title="World Wide Web">
    Web</a> or within a closed system, to generate more relevant results.
```

and edges link each term to its following terms within a sliding window of size 3, in order to capture context. They also defined the TW-IDF retrieval model, based on retrieval heuristics (Fang, Tao, and Zhai [2004]), over the graph representation, and built on the indegree of the nodes. The idea was to measure the "number of contexts" a given term appears in. In their implementation, the graph-of-word was generated per document, computing the TW metric and storing it within an inverted file instead of the typical TF, and so the graph structures for each document were discarded. They evaluated their TW-IDF weighting function with and without regularization over document length, as well as with and without parameter tuning for the pivoted document length normalization b parameter. They did a comparison of TW-IDF with TF-IDF (Salton and McGill [1986]) and BM25 (Robertson et al. [1994]), as well as Piv+ and BM25+ (Lv and Zhai [2011]), showing that TW-IDF consistently outperformed the other weighting functions, particularly in realistic conditions, where parameter tuning is costly and is seldom an option.

Document Representation

In order to illustrate how document representation is done in the graph-of-word model, we first introduce an example document based on the first sentence of the Wikipedia page for "Semantic Search", as shown in Listing 3.3. The resulting graph-of-word for the document, assuming a sliding window of 3 that covers the analyzed term, is illustrated in Figure 3.1. As we can see, words were converted to lowercase and stopwords were removed. Working under the assumption that the terms following a given term would be a good representation of context, the indegree of this graph is a measurement of term relevance that takes term dependence into consideration.

The authors originally build this graph for each document, extracting the term weight (TW) from the indegree of each node and storing the value in the inverted index, discarding the graph structure. Instead of following this implementation approach, we stored each graph in a Neo4j[3] graph database, using a `name` property with the term adding a `doc_id` property to each edge, in order to enable the indegree computation for any document-induced subgraph. This incurs in an obvious cost in efficiency that is justified only by the flexibility to explore and extend the graph, as well as the ability to produce graph computations using the Gremlin DSL. By definition, a graph database is a graph storage and retrieval system that provides index-free adjacency, that is, each node can access any of its neighbors without first querying an index. The Neo4j instances we use throughout this work are configured with node and relationship (edge) auto indexing. This means that nodes and edges can be reached or filtered through their properties using an index, but traversals are still index-free. In fact, Gremlin graphs follow the property graph model, which can be formally defined as a directed, binary, labeled, attributed multigraph. This means that relationships are binary (i.e., it's not a hypergraph), $(v_1, v_2) \neq (v_2, v_1) : \forall v_1, v_2 \in V$ (i.e., direction matters in defining a relationship), nodes and edges have types (i.e., the graph is labeled), and nodes and edges can have properties (i.e., the graph is attributed).

Document Retrieval

Document retrieval consists of computing the TW-IDF for each term in the query, over each document in the collection. Next, we will present a short summary of the most relevant techniques used by Rousseau

[3]https://neo4j.com/

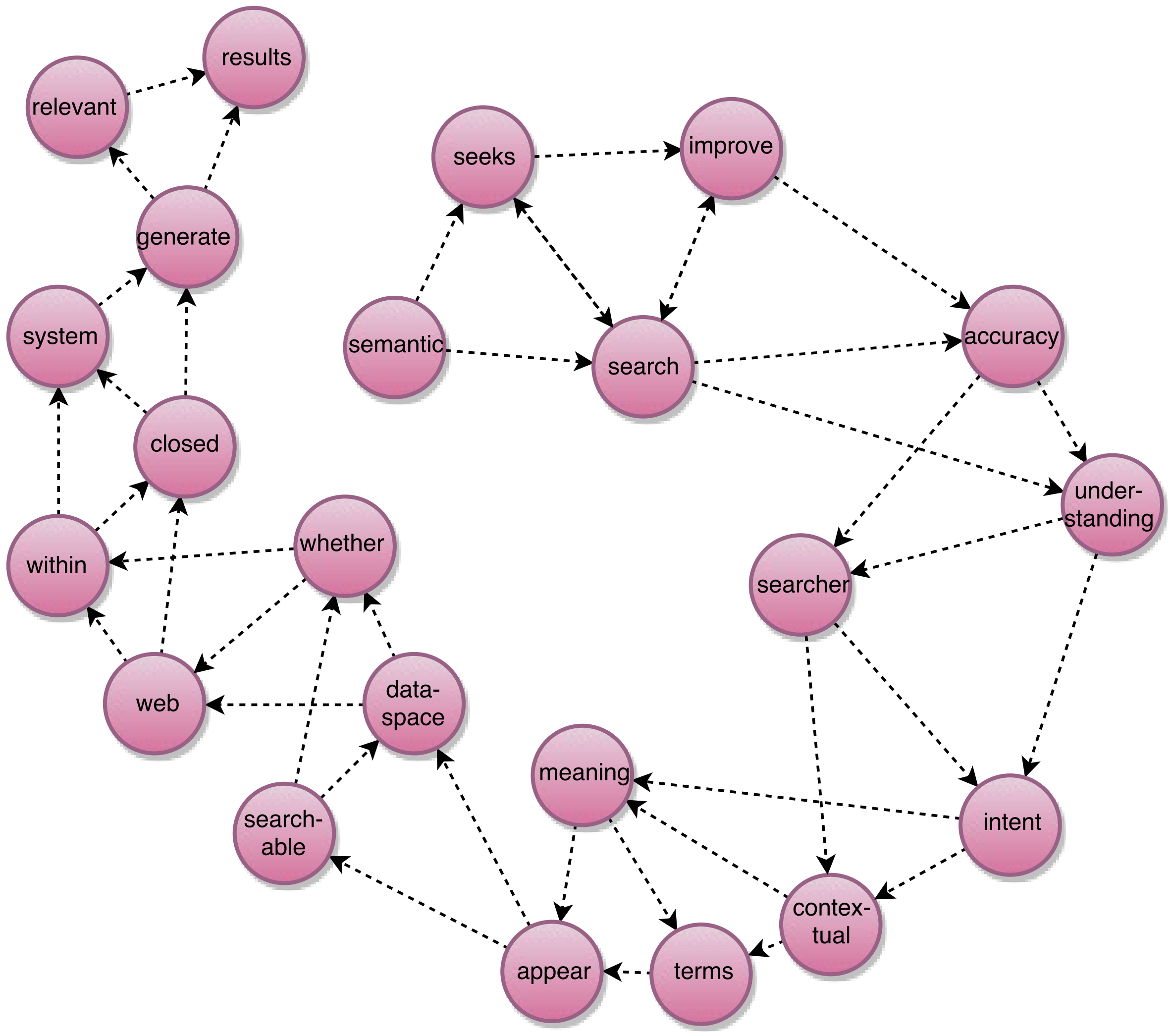


Figure 3.1: Graph-of-word document representation model. Nodes represent terms, while arrows link each term to the following two terms (within a sliding window of size three). This aims at capturing context.

and Vazirgiannis (2013) to construct their retrieval model. The TW-IDF weighting function is displayed in Equation 3.1.

$$TW\text{-}IDF(t,d) = \frac{tw(t,d)}{1 - b + b \times \frac{|d|}{avdl}} \times log\frac{N+1}{df(t)} \tag{3.1}$$

The authors took advantage of information retrieval heuristics, as formally studied by Fang, Tao, and Zhai (2004) and, later, by Lv and Zhai (2011), to obtain a valid weighting function. These heuristics form a toolkit of functions that can either replace TF or be combined through composition, in order to obtain many of the well-known weighting schemes, such as TF-IDF (vector space model), BM25 (divergence from randomness) or Dirichlet prior smoothing (language modeling). Table 3.1 presents an overview, based on the description by Rousseau and Vazirgiannis (2013), on information retrieval heuristics, and Table 3.2 describes each individual component used within the functions. TW-IDF was obtained by applying pivoted document length normalization to the term weight (TW) a multiplying it by the inverse document frequency (IDF).

The document retrieval process, over the graph database implementation, requires three arguments:

Table 3.1: Overview on information retrieval heuristics.

Designation	Goal	Functions
Concave function	To decrease the marginal gain of multiple occurrences of a term within a particular document.	$TF_k(t,d) = \frac{(k_1+1) \times tf(t,d)}{k_1 + t_f(t,d)}$ $TF_l(t,d) = 1 + ln[1 + ln[tf(t,d)]]$
Pivoted length normalization	To scale by document length, handling documents of varying lengths and ensuring "probability of retrieval matches probability of relevance".	$TF_p(t,d) = \frac{tf(t,d)}{1 - b + b \times \frac{\lvert d \rvert}{avdl}}$
Lower-bounding regularization	To scale by document length, handling documents of varying lengths and ensuring "probability of retrieval matches probability of relevance".	$TF_\delta = \begin{cases} tf(t,d) + \delta & \text{if } tf(t,d) > 0 \\ & \text{otherwise} \end{cases}$

an array of query tokens, and an offset and a limit for pagination purposes. In order to calculate TW-IDF, we follow the subsequent steps:

1. For each node matching a query token, we count the number of incoming edges per document, obtaining the $tw(t, d)$ component.
2. For each node matching a query token, we count the number of unique `doc_id` for incoming and outgoing edges, obtaining the document frequency $df(t)$ component.
3. We compute the length $|d|$ of each document, based on the number of unique terms per document, by grouping edges by `doc_id` and counting the number of target nodes. This deviates slightly from the original graph-of-word model, as a restriction imposed by our current implementation, which we plan to correct in the future, by adding an edge weight and summing over this weight instead of counting.
4. Based on the document lengths computed in the previous step, we calculate the *avdl* average document length.
5. Corpus size can be computed by counting the number of unique `doc_id` over all edges.
6. Finally, we iterate over the precomputed indegree of each term node and calculate the $tw\text{-}idf(t, d)$ for each document. We then group by document and sum the individual $tw\text{-}idf(t, d)$ values to obtain the final score for the document.

The actual Gremlin code used to query the graph-of-word is available in Appendix A.1.

Table 3.2: Description of individual components used in information retrieval heuristic functions.

Component	Description
$tf(t, d)$	Term frequency of the term t in the document d.
k_1	Asymptotical maximal gain achievable by multiple occurrences compared to a single occurrence (by default, $k_1 = 1.2$).
$b \in [0,1]$	Slope parameter of the tilting.
$\lvert d \rvert$	Document length.
$avdl$	Average document length across the corpus.
δ	Lower-bounding gap (by default, $\delta = 1.0$).

Graph of Entity

We propose the graph-of-entity as an extension to the graph-of-word with the goal of unifying all available information sources (text and knowledge) in a single data structure. The intuition is that seamlessly integrating text and knowledge has the potential to unlock novel solutions for multiple tasks within entity-oriented search. Furthermore, we believe as well that we might be able to somehow combine these solutions in order to improve the outcome of each other and the overall performance of the system.

Information retrieval (IR) and information extraction (IE) have always been two complementing areas, but in the last few years we have seen an increasing combination of the two areas in hybrid systems that are not clearly an IR or IE system, but rather both. This is the case of entity-oriented search engines like Google. The goal is to use the best available information, which sometimes means taking advantage of a well-curated knowledge base, automatically building our own or simply retrieving complete textual documents or specific relevant passages in an attempt to solve the information needs of the users.

There are several tasks to be tackled in the context of a search engine, well beyond document representation and indexing or document ranking and retrieval. In particular, when there is additional knowledge available, and given the relevance of entities in search queries (Bautin and Skiena [2007]), we can process the query in order to better understand user intent (Hu et al. [2009]; Rose and Levinson [2004]; Li and Xu [2014]; Hasibi, Balog, and Bratsberg [2015]; Gupta and Bendersky [2015]; Foley, O'Connor, and Allan [2016]; Pound et al. [2012]; Tan et al. [2017]), taking advantage of query entity linking to improve retrieval.

Our proposal is that we bridge text and knowledge through a graph-based data structure to improve entity-oriented search. As proof of concept, we implement query entity linking over the graph, weighting candidate entities based on the edges between text and entity nodes and integrating these weights into the ranking function. We view the tasks in entity-oriented search as a whole, proposing that, instead of choosing the best query segmentation and semantic tags, we should propagate uncertainty and combine it with the ranking function as an additional feature — a decision on relevance should only be taken based on maximal information.

Next, we describe the graph-of-entity model, explaining how it evolved from the graph-of-word, and working as a first approach to the problem of unifying text and knowledge for better retrieval.

Document Representation

Let us assume that a document consists of a unique identifier, a text field and a list of triples containing knowledge related to the covered topic(s). Figure 3.2 shows the graph-of-entity for the same example document from Listing 3.3, that was also used to illustrate the graph-of-word in Section 3.2. This is a labeled multigraph, with two node types — `term`, in pink, and `entity`, in green — and three edge types — `related_to`, represented as a solid arrow between two entity nodes, `before`, represented as a dashed arrow between two term nodes, and `contained_in`, represented as a dotted arrow between a term node and an entity node. Perhaps the biggest difference between the graph-of-entity and the graph-of-word, apart from the extensions, is that we do not use a sliding window to obtain the edges and establish a context. Instead, we simply capture term dependence by modeling term sequence. The same information is captured within this graph and we can obtain the same statistic by traversing all paths from a given node and under a maximum distance which is equivalent to the window size. The impact in storage size is visible, as we will show in Section 3.4.1, and it allows for alternative techniques like using random walks instead of traversing all paths to compute probabilities for a weighted context, introducing yet another level of uncertainty that can be propagated to the ranking function.

The Neo4j graph database for the graph-of-entity stores a `doc_id` property associated with `before` edges, as well as with searchable `entity` nodes. Some metadata is also stored within `entity` nodes, including `name` and `url` properties, when available. Similarly to the graph-of-word implementation, we configured node and relationship (edge) auto indexing for the graph database.

Document Retrieval

We model document retrieval as an entity retrieval problem, proposing an $EW\text{-}TEF(Q, e)$ weighting function where we take into consideration the entity weight (EW), computed from coverage (fraction

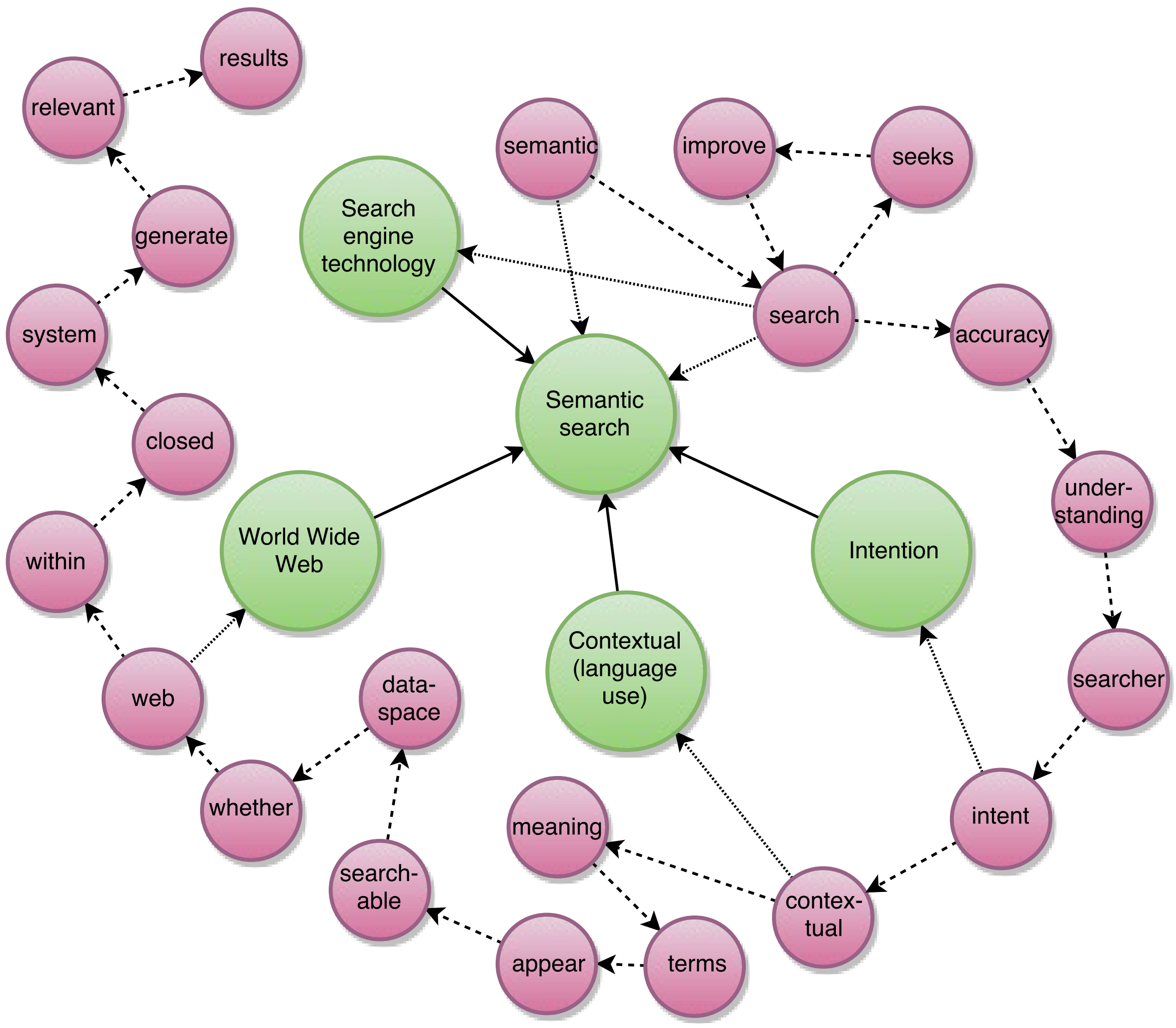


Figure 3.2: Graph-of-entity document representation model. Solid arrows represent `related_to` relations between entity nodes (in green), dashed arrows represent `before` relations between term nodes (in pink), and dotted arrows represent `contained_in` relations between term nodes and entity nodes.

of links to seed entities) and average weighted inverse path length (from each entity to seed entities, weighted by the seed node confidence). We then combine the results obtained from entity weight with the results obtained from a term entity frequency (TEF) component. Seed entities are obtained through a query entity linking process, consisting of all entities with a given probability of representing the query. While the ranking function for the graph-of-entity is far from being publishable, given it still requires further research and improvements, it is already worth documenting and evaluating, in order to establish a baseline that we can build upon. It is of particular interest to discuss the unification of query entity linking and entity ranking as a single task, in a system that is not purely based on either text or knowledge, heavily relying, instead, on combining weights for terms and entities.

Next, we will describe each computational step taken towards obtaining a particular component of the final score, starting with query entity linking to obtain a set of seed nodes, then computing an entity weight for each entity in the graph with an associated `doc_id` and extending the results list with any missing text-based results. The biggest pitfall of the final step is that it uses the standard boolean model, meaning that a document is either included in the results list when it's relevant or not at all when it's not relevant, but it is not ranked, having a score of zero. This is clearly an area that requires further work, in particular regarding the combination of EW and TEF. However, we included this component simply to improve recall, ensuring that graph-of-entity is at least on par with graph-of-word.

Query Entity Linking

The retrieval process begins with the identification of the term nodes corresponding to each query term. The selection process of seed nodes is then analogous to query entity linking, with two main differences: (*i*) linking is probabilistic instead of deterministic, and (*ii*) whenever a term cannot be linked with an entity, the term node is used as a seed node with unitary weight, in a zero to one scale. The process is then described as follows:

1. For each node matching a query token (from now on, query node), follow all `contained_in` edges and gather entity nodes along with query nodes that have no outgoing `contained_in` edges. These are the seed nodes.

2. For each seed node, either return a confidence weight of one, if it is a term node, or the fraction of query nodes linking to the entity seed node per total number of incoming `contained_in` edges.

For further details on the implementation of the query entity linking process, pay special attention to the `seedScoresPipe` and `seedScores` variables, as displayed in Appendix A.2.

Entity Weighting

The goal is then to rank all entity nodes based on the identified seed nodes that, in turn, are a fuzzy representation of the query in the graph. We compute the relevance of an entity in regards to the seed nodes based on the geodesic distance. Particularly, we consider that the further away an entity node is from the seed nodes, the less relevant it is to the query — as an indicator or relevance, we use the inverse of the distance between an entity node and a seed node, under a given maximum distance threshold. We then multiply this by the confidence weight of each seed node — an entity might be relevant for a seed node, but if that seed node is only weakly linked to the query, then it is less relevant (unless no other, more relevant results are available). We call this metric the average weighted inverse distance to seed nodes.

The second component of the entity weight that we compute is the coverage of an entity node in regards to the seed nodes — the more seed nodes are linked (i.e., have a path) to an entity node, the more relevant they are to the query. The coverage is simply the fraction of seeds with a path to the entity per total number of seeds. The product of the coverage and the average weighted inverse distance to seed nodes is the entity weight (EW).

For further details on the implementation of the entity weight, pay special attention to the `distances-ToSeedsPerEntity` variable, as well as the first block of code within the `ewTef` variable, as displayed in Appendix A.2.

Factoring in Text

Finally, after directly using the entity weight as our ranking function and experimenting with Army ANT, we noticed that the graph-of-word would, in some cases, return results when none were returned by the graph-of-entity. At this point, we decided to introduce the term entity frequency (TEF) component. The idea was to match the `doc_id` property, found within `before` edges between a query node and another term node, with the respective entity nodes (with the same `doc_id`), calculating the number of entities per term (i.e., the term entity frequency).

While we experimented with log normalization for TEF, due to time constraints we were unable to find a good candidate ranking function to combine EW and TEF. Thus, we simply added all documents retrieved via TEF that hadn't already been retrieved by EW, giving them a zero score. We did this simply to improve recall and assuming a continued improvement of the graph-of-entity ranking function.

For further details on the implementation of the entity weight, pay special attention to the `term-EntityFrequency` variable, as well as the second block of code within the `ewTef` variable, as displayed in Appendix A.2.

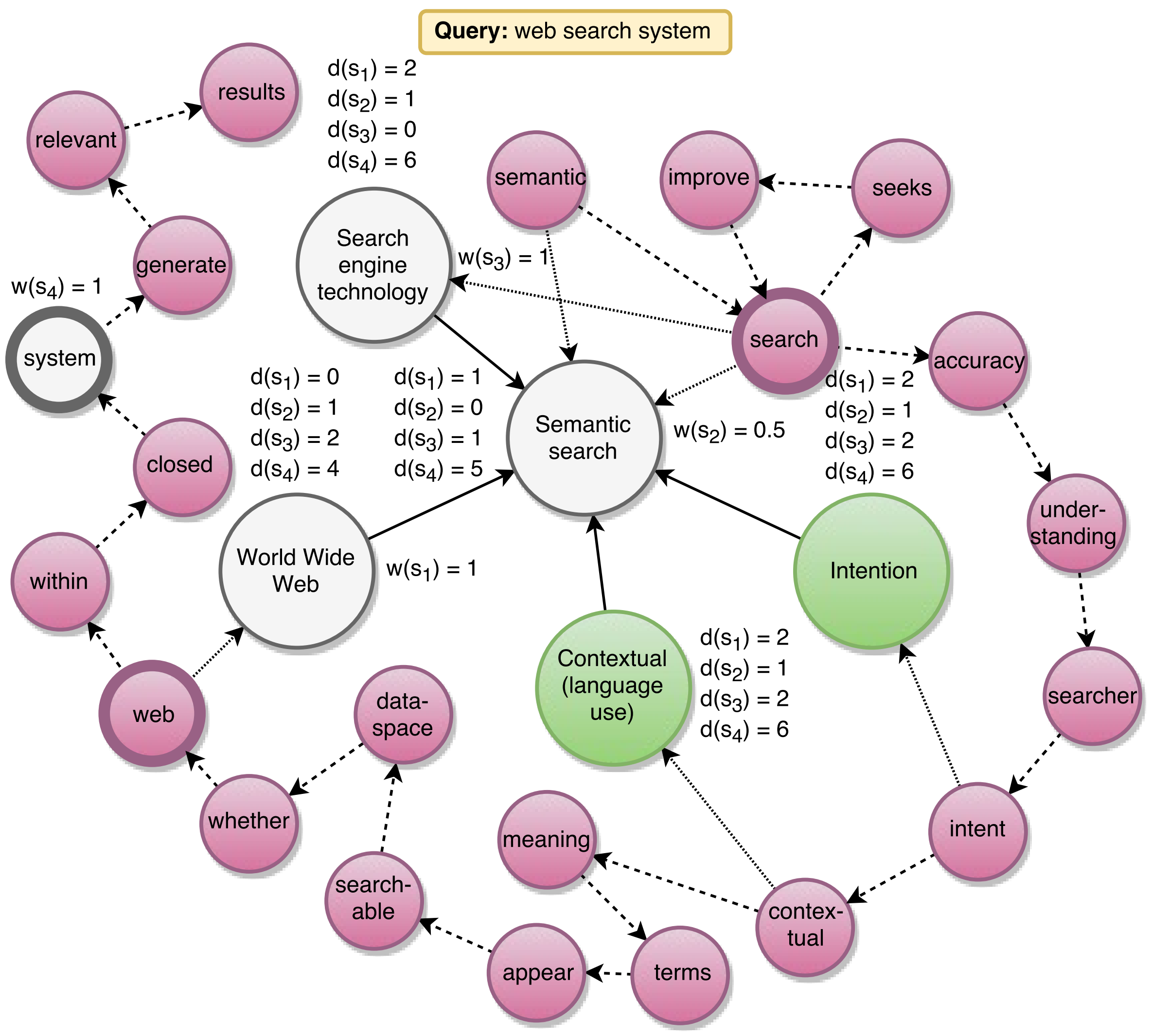


Figure 3.3: Searching for `web search system` over the graph-of-entity. Thicker node borders indicate query nodes, while light gray nodes correspond to identified seed nodes. We also show the confidence weight $w(s)$ for each seed node, as well as the distances $d(s)$ from seed nodes to each entity node.

Practical Example

Let us assume we issue the query `web search system` over the graph-of-entity for the example document in Listing 3.3. Notice also that the graph-of-entity does not exist in isolation for each document, but instead everything is connected to represent the document collection. However, for this practical example, we only use the graph-of-entity for the first paragraph of the "Semantic Search" Wikipedia page.

As we can see in Figure 3.3, the first step was to identify the term nodes matching the query terms. These are displayed with a thicker border. We then identify the seed nodes that will represent the query in the graph. These are displayed in light gray. There are three entity seed nodes — `World Wide Web`, `Semantic Search` and `Search engine technology` — and one term seed node — `system`. For each seed node, we annotate the graph with the confidence weight. Both the `World Wide Web`, `Search engine technology` and `system` have a maximum confidence weight of one, while `Semantic Search` has a confidence weight of 0.5, since only one in two `contained_in` incoming edges comes from a query node (`search`). We then calculate, for each entity node, the distance from each seed. In this example, given we are analyzing a single document, there is a path from every seed to every entity node, but, in the event there wasn't, some of these distances would be infinite (undefined). The coverage for each entity node in this graph is maximal, with a value of one, since there is always a path from an entity

Table 3.3: Entity weight for all entities in the graph-of-entity from Figure 3.3.

Rank	**Entity**	**EW(Q, e)**
1	World Wide Web	0.5944444
2	Search engine technology	0.5753968
3	Semantic search	0.5555556
4	Contextual (language use)	0.3531746
5	Intention	0.3531746

node to all seed nodes. Given all these values, we can easily calculate the entity weight for each entity e and query Q. Individual entity weight values are shown in Table 3.3, as calculated using Equation 3.2 and already ranked accordingly. The coverage is always one, which multiples with the average of the weighted inverted distances.

$$EW(Q,e) = 1 \cdot \frac{1}{3} \cdot \left(w(s_1) \cdot \frac{1}{1+d(s_1)} + w(s_2) \cdot \frac{1}{1+d(s_2)} + w(s_3) \cdot \frac{1}{1+d(s_3)} + w(s_4) \cdot \frac{1}{1+d(s_4)} \right) \quad (3.2)$$

Brief Discussion From the elaboration of this practical example, we found three interesting details. First, we noticed there was an error in the calculation of the inverse distance to seed nodes as, in some cases, a division by zero was possible. We corrected this by using $\frac{2}{1+d}$ instead. Then, regarding the document retrieval strategy, we found that an entity that is a seed node and often a good candidate for a high score, was potentially being ignore when there wasn't a path from other seed nodes. While, in practice, we didn't notice the impact of this, it is something that we must correct in the future. Finally, we also noticed that, while the weight of a term seed node should be higher than that of an entity seed node, simply because there usually are more entity seed nodes, we might have exaggerated in setting the weight of a term seed node to one. In the future, we should experiment with lower fixed values or even a function instead.

Evaluation

In this section, we present some performance statistics regarding the current indexing efficiency of the rather underoptimized implementation based on graph databases. We also present disk space and graph statistics, comparing the indexes for Wikipedia Relation Extraction Data v1.0 and for INEX 2009 Collection. We also present the MAP and NCDG@10 effectiveness metrics for each graph-based retrieval model, computed using the Army ANT workbench over a small sample of the INEX 2009 Collection.

Index Performance Statistics

During the preparation of this implementation, we opted to not prematurely optimize the system, as our focus on on effectiveness. Otherwise, we would have reconsidered a different approach to implement the index, given that a graph database already uses two inverted indexes, for the node and edge properties, besides the data structure for index-free adjacency graph traversals. Nevertheless, in Tables 3.4 and 3.5, we present several statistics, per dataset, regarding Neo4j and MongoDB storage of both the graph-of-word and graph-of-entity models. As we can see, the graph-of-entity takes longer to be built, but also results in a smaller index. When comparing the two index time for either dataset, it is clear that the current approach does not scale well and even after basic optimizations using the `BulkLoaderVertexProgram`, we predict that it still won't be up to par on scalability (at least not without a different storage strategy).

As we can see, the storage space taken by the metadata is minimal — it's always under 1 MB. For the Wikipedia Relation Extraction Data v1.0, the resulting graph-of-word contained over 7k nodes and 73k edges, while the graph-of-entity contained over 14k nodes and 49k edges, resulting in double the nodes and nearly half the edges. For the INEX 2009 Collection, the resulting graph-of-word contained over 135k nodes and 2.4 million edges, while the graph-of-entity contained over 284k nodes and 1 million edges, resulting in less than double the nodes and less than half the edges.

Table 3.4: Index and database statistics for Wikipedia Relation Extraction Data v1.0.

Neo4j Data	
Index Time	2m44s
Index Size	143 MB
Number of Vertices	7,957
Number of Edges	73,232
Number of Documents	257
MongoDB Data	
Storage Size	0.1757 MB
Index Size	0.0068 MB

(a) Graph of Word.

Neo4j Data	
Index Time	32m41s
Index Size	40 MB
Number of Vertices	14,551
Number of Edges	49,095
Number of Documents	257
MongoDB Data	
Storage Size	1.0078 MB
Index Size	0.0849 MB

(b) Graph of Entity.

Table 3.5: Index and database statistics for INEX 2009 Collection (subset containing `pages/000`, `pages/001`, `pages/002` and `pages/003` from the `pages25.tar.bz2` archive).

Neo4j Data	
Index Time	52h58m
Index Size	1.4 GB
Number of Vertices	135,005
Number of Edges	2,363,741
Number of Documents	2608
MongoDB Data	
Storage Size	0.6758 MB
Index Size	0.0936 MB

(a) Graph of Word.

Neo4j Data	
Index Time	67h55m
Index Size	823 MB
Number of Vertices	284,902
Number of Edges	1,014,884
Number of Documents	2610
MongoDB Data	
Storage Size	0.6758 MB
Index Size	0.0936 MB

(b) Graph of Entity.

Retrieval Model Effectiveness

We measured the retrieval model effectiveness based on the mean average precision (MAP) and the normalized discounted cumulative gain for the top 10 results (NDCG@10). We used the topics and relevance judgments provided along with the INEX 2009 Collection to assess both models. However, given the inefficiency of our implementation, we were only able to test the system based on a small subset of nearly three thousand documents. This was not ideal since, as we verified, many of the queries for each topic did not return any results, given only a small fraction of the document collection was indexed.

Nevertheless, we present the results that we obtained for the MAP and NDCG@10 metric for either model. In the future, we intent to improve the indexing performance, in order to be able to index the complete dataset and properly rerun the evaluation tasks in Army ANT. The relevance judgment files present an assessment for the context of passage retrieval, thus providing the number of relevant characters in a document, for a given topic, as the judgment grade. For the MAP metric, we only required a binary relevance grade and thus simply considered that zero relevant characters meant the document was not relevant and vice-versa. For the NDCG@10, it is more common to use a grade between 0 and 3

Table 3.6: Evaluation metrics for the graph-of-word and graph-of-entity retrieval models over a small sample of the INEX 2009 Collection.

Retrieval Model	**MAP**	**NDCG@10**
Graph of Word	0.0055	0.0015
Graph of Entity	0.0048	0.0061

to calculate the metric. We experimented with the number of relevant characters instead, but it resulted in infinitesimal values that could only read as zero. An alternative would be to calculate the fraction of relevant characters instead and then split the space into intervals with an equal number of documents, assigning a grade from 0 to 3 based on the prepared bins. Given the performance issues related to the processing of this moderately large dataset, we decided to use another alternative, which consisted on simply assigning a binary grade as we did for the MAP metric. Table 3.6 shows the effectiveness metrics obtained for both models. As we can see, the metrics do not agree, with MAP being slightly larger for the graph-of-word and NDCG@10 being slightly larger for the graph-of-entity. After a brief analysis of the intermediate file for MAP, with the individual average precisions for each topic, we found that the graph-of-word only returned results for 5 out of 52 topics, while the graph-of-entity only returned results for 1 out of 52 topics. Clearly this is not enough to support any conclusion, but instead we take it as an opportunity to illustrate the evaluation methodology we have already prepared for future research in the area.

Chapter 4

Conclusion

We have presented the Army ANT workbench for innovation in entity-oriented search. This software package has been created as a tool for research and education, with the specific problem of evaluating graph-based entity-oriented retrieval models and to support the incoming participation in the TREC 2017 Open Search track.

We began by introducing TREC and the Open Search track, along with our proposed methodology for innovation in the area of entity-oriented search and a bit of context as to the reason of exploring graph-based retrieval models with the goal of unifying information sources and retrieval tasks in a single model. We then presented the Army ANT system architecture, describing how it can be extended by other researchers to implemented and evaluate their own models. We also presented the user's manual for the workbench, where we described the requirements and installation procedure, along with the command line and web interfaces. Finally, we experimented with two different graph-based models: graph-of-word, an already existing model by Rousseau and Vazirgiannis (2013), and graph-of-entity, our proposal of a model capable of unifying text and knowledge and of supporting the integration of query entity linking and entity ranking by propagating uncertainty and calculating a combined final score. We described the two datasets used for the development and testing of the system, as well as for evaluation of the assessed models. We closed by presenting some index performance statistics, along with the evaluation of the retrieval model effectiveness based on MAP and NDCG@10. Given we only used a subset of the INEX 2009 Collection, due to performance restrictions, the final result was inconclusive, based on the low support given only by a few topics with results.

Future Work

As future work, we propose to improve the graph database loading process, in order to index the full INEX 2009 Collection and to conveniently measure effectiveness with MAP and NDCG@10. An alternative, although we argue premature, would be to work on a custom, more efficient, graph-based index data structure. Another line of research that will be the focus of this work in the future is the improvement of the weighting scheme for graph-of-entity and, in particular, a better integration of the term entity frequency (TEF) component, which currently only includes relevant documents with a score of zero.

An intuition we gained based on the analysis of the diagram for the graph-of-entity, when thinking about measuring the "number of contexts" for a term, similarly to what we do for the graph-of-word, was that we might use random walks instead of traversing all paths, in order to compute some kind of weighted context; this is also an interesting path to pursue.

Finally, we also done quite an extensive review on graph metrics (Hernández and Van Mieghem [2011]), graph substructures and traversal algorithms that can be useful in the design of graph-based retrieval strategies. Some of the concepts we analyzed included: several node-centric metrics, for instance centralities — degree, closeness, random walk closeness, betweenness, eigenvector, PageRank —, as well as concepts like eccentricity, resistance distance, local clustering coefficient, coreness, rich-club coefficient, or vertex-connectivity. We also considered the behavior of such metrics in a multidimensional network (Magnani et al. [2013]), where we identified at least the degree per dimension, distances within a subset of dimensions, multidimensional betweenness centrality (based on the shortest paths from a subset of dimensions) and dimension relevance. Graph-wide metrics, like conductance, were also considered, as well as graph clustering techniques, commonly known as community detection, because of their popularity in social network analysis. While there is a well-defined intuition behind each of these concepts and usually it is relatively simple to describe the real-world representation of a given metric or subgraph within a particular graph, there is a particularly interesting technique that has not, to our knowledge, been studied in-depth within network science, which is the quantum walk in a graph. We would like to experiment with quantum walks, in order to understand their effect on the graph-of-entity, imputing some kind of semantic to this traversal strategy. For instance, random walks are useful to compute probabilities that can be used to detect community structure. What about quantum walks?

Finally, and perhaps more importantly in the context of this assignment, we will participate in the TREC 2017 Open Search track, where we will assess the effectiveness of graph-of-word and graph-of-entity based on their Living Labs platform. This has always been short term the goal of building Army ANT and proposing the graph-of-entity. With all the work we describe in this report, we believe we are well set — with knowledge, workbenches and goals — to innovate entity-oriented search and give a relevant contribution to the overall area of search in information retrieval.

Acknowledgments

I would like to thank Carla Lopes for all the useful discussions we had during the course of this work and, in particular, regarding information retrieval evaluation metrics. I would also like to thank Sérgio Nunes for the many discussions we had during our meetings to discuss the PhD or to supervise master's students, which greatly contributed to the creation of Army ANT as an research and educational platform. Finally, I would like to thank Tiago Devezas for suggesting the use of the Spectre.css[1] toolkit to develop the front-end for Army ANT and Nelson Pereira for suggesting draw.io[2], which I have used to illustrate the graph-based retrieval models.

[1]`https://picturepan2.github.io/spectre/`
[2]`https://www.draw.io/`

Bibliography

[1] Krisztian Balog, Anne Schuth, N Tavakolpoursaleh, P Schaer, PY Chuang, J Wu, and CL Giles. "Overview of the trec 2016 open search track". In: *Proceedings of the Twenty-Fifth Text REtrieval Conference (TREC 2016). NIST*. 2016.

[2] Hannah Bast, Björn Buchhold, Elmar Haussmann, et al. "Semantic Search on Text and Knowledge Bases". In: *Foundations and Trends® in Information Retrieval* 10.2-3 (2016), pp. 119–271.

[3] Mikhail Bautin and Steven Skiena. "Concordance-Based Entity-Oriented Search". In: *The 2007 IEEE / WIC / ACM Conference on Web Intelligence (WI '07)*. 2007, pp. 2–5.

[4] Aron Culotta, Andrew McCallum, and Jonathan Betz. "Integrating Probabilistic Extraction Models and Data Mining to Discover Relations and Patterns in Text". In: *Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics, Proceedings, June 4-9, 2006, New York, New York, USA*. 2006. URL: `http://aclweb.org/anthology/N/N06/N06-1038.pdf`.

[5] José Luís Devezas, Sérgio Nunes, and Cristina Ribeiro. "FEUP at TREC 2010 Blog Track: Using h-index for blog ranking". In: *NIST Special Publication* (2010).

[6] P. Domingos. *The Master Algorithm: How the Quest for the Ultimate Learning Machine Will Remake Our World*. Basic Books. Basic Books, 2015. ISBN: 9780465065707. URL: `https://books.google.pt/books?id=glUtrgEACAAJ`.

[7] Christos Faloutsos and Stavros Christodoulakis. "Signature Files: An Access Method for Documents and Its Analytical Performance Evaluation". In: *ACM Trans. Inf. Syst.* 2 (1984), pp. 267–288.

[8] Hui Fang, Tao Tao, and ChengXiang Zhai. "A formal study of information retrieval heuristics". In: *SIGIR 2004: Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Sheffield, UK, July 25-29, 2004*. 2004, pp. 49–56. DOI: `10.1145/1008992.1009004`. URL: `http://doi.acm.org/10.1145/1008992.1009004`.

[9] John Foley, Brendan O'Connor, and James Allan. "Improving Entity Ranking for Keyword Queries". In: *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management, CIKM 2016, Indianapolis, IN, USA, October 24-28, 2016*. 2016, pp. 2061–2064. DOI: `10.1145/2983323.2983909`. URL: `http://doi.acm.org/10.1145/2983323.2983909`.

[10] Manish Gupta and Michael Bendersky. "Information Retrieval with Verbose Queries". In: *Foundations and Trends in Information Retrieval* 9.3-4 (2015), pp. 91–208. DOI: `10.1561/1500000050`. URL: `https://doi.org/10.1561/1500000050`.

[11] Faegheh Hasibi, Krisztian Balog, and Svein Erik Bratsberg. "Entity Linking in Queries: Tasks and Evaluation". In: *Proceedings of the 2015 International Conference on The Theory of Information Retrieval, ICTIR 2015, Northampton, Massachusetts, USA, September 27-30, 2015*. 2015, pp. 171–180. DOI: `10.1145/2808194.2809473`. URL: `http://doi.acm.org/10.1145/2808194.2809473`.

[12] Javier Martın Hernández and Piet Van Mieghem. "Classification of graph metrics". In: *Delft University of Technology, Tech. Rep* (2011).

[13] Jian Hu, Gang Wang, Fred Lochovsky, Jian-tao Sun, and Zheng Chen. "Understanding user's query intent with wikipedia". In: *Proceedings of the 18th international conference on World wide web*. ACM. 2009, pp. 471–480.

[14] Alfred Inselberg. "The plane with parallel coordinates". In: *The Visual Computer* 1.2 (1985), pp. 69–91. DOI: `10.1007/BF01898350`. URL: `https://doi.org/10.1007/BF01898350`.

[15] Hang Li and Jun Xu. "Semantic Matching in Search". In: *Foundations and Trends in Information Retrieval* 7.5 (2014), pp. 343–469. DOI: 10.1561/1500000035. URL: https://doi.org/10.1561/1500000035.

[16] Yuanhua Lv and ChengXiang Zhai. "Lower-bounding term frequency normalization". In: *Proceedings of the 20th ACM Conference on Information and Knowledge Management, CIKM 2011, Glasgow, United Kingdom, October 24-28, 2011.* 2011, pp. 7–16. DOI: 10.1145/2063576.2063584. URL: http://doi.acm.org/10.1145/2063576.2063584.

[17] MARGGF Magnani, Anna Monreale, Giulio Rossetti, and Fosca Giannotti. "On multidimensional network measures". In: *Italian conference on Sistemi Evoluti per le Basi di Dati (SEBD).* 2013.

[18] Udi Manber and Gene Myers. "Suffix Arrays: A New Method for On-Line String Searches". In: *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1990, San Francisco, California.* 1990, pp. 319–327. URL: http://dl.acm.org/citation.cfm?id=320176.320218.

[19] Christopher D Manning, Prabhakar Raghavan, Hinrich Schütze, et al. *Introduction to information retrieval.* Vol. 1. 1. Cambridge university press Cambridge, 2008.

[20] Andrea Moro, Alessandro Raganato, and Roberto Navigli. "Entity Linking meets Word Sense Disambiguation: a Unified Approach". In: *TACL* 2 (2014), pp. 231–244. URL: https://tacl2013.cs.columbia.edu/ojs/index.php/tacl/article/view/291.

[21] Jeffrey Pound, Alexander K Hudek, Ihab F Ilyas, and Grant Weddell. "Interpreting keyword queries over web knowledge bases". In: *Proceedings of the 21st ACM international conference on Information and knowledge management.* ACM. 2012, pp. 305–314.

[22] Stephen E. Robertson, Steve Walker, Susan Jones, Micheline Hancock-Beaulieu, and Mike Gatford. "Okapi at TREC-3". In: *TREC.* 1994.

[23] Daniel E. Rose and Danny Levinson. "Understanding user goals in web search". In: *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004.* 2004, pp. 13–19. DOI: 10.1145/988672.988675. URL: http://doi.acm.org/10.1145/988672.988675.

[24] François Rousseau and Michalis Vazirgiannis. "Graph-of-word and TW-IDF: new approach to ad hoc IR". In: *Proceedings of the 22nd ACM international conference on Information & Knowledge Management.* ACM. 2013, pp. 59–68.

[25] Gerard Salton and Michael J McGill. "Introduction to modern information retrieval". In: (1986).

[26] Ralf Schenkel, Fabian M. Suchanek, and Gjergji Kasneci. "YAWN: A Semantically Annotated Wikipedia XML Corpus". In: *Datenbanksysteme in Business, Technologie und Web (BTW 2007), 12. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), Proceedings, 7.-9. März 2007, Aachen, Germany.* 2007, pp. 277–291. URL: http://subs.emis.de/LNI/Proceedings/Proceedings103/article1404.html.

[27] Chuanqi Tan, Furu Wei, Pengjie Ren, Weifeng Lv, and Ming Zhou. "Entity Linking for Queries by Searching Wikipedia Sentences". In: *arXiv preprint arXiv:1704.02788* (2017).

[28] Ellen M. Voorhees. "The Efficiency of Inverted Index and Cluster Searches". In: *SIGIR'86, Proceedings of the 9th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Pisa, Italy, September 8-10, 1986.* 1986, pp. 164–174. DOI: 10.1145/253168.253203. URL: http://doi.acm.org/10.1145/253168.253203.

[29] Justin Zobel and Alistair Moffat. "Inverted files for text search engines". In: *ACM computing surveys (CSUR)* 38.2 (2006), p. 6.

[30] Justin Zobel, Alistair Moffat, and Kotagiri Ramamohanarao. "Inverted Files Versus Signature Files for Text Indexing". In: *ACM Trans. Database Syst.* 23.4 (1998), pp. 453–490. DOI: 10.1145/296854.277632. URL: http://doi.acm.org/10.1145/296854.277632.

Appendix A

Code Listings

Graph of Word

```
/**
 * Calculate the TW-IDF, from Graph of Word model, for a term in a document.
 *
 * @param indegree Term vertice indegree.
 * @param docFreq Document frequency of the term. The number of documents containing the term.
 * @param docLength The number of characters of the document.
 * @param avgDocLength The average number of characters of the documents in the corpus.
 * @param corpusSize The number of documents in the corpus.
 * @param b The slope parameter of the tilting. Fixed at 0.003 for TW-IDF.
 */
def twIdf(indegree, docFreq, docLength, avgDocLength, corpusSize, b=0.003) {
  indegree / (1 - b + b * docLength / avgDocLength) * Math.log((corpusSize + 1) / docFreq)
}

//queryTokens = ['born', 'new', 'york']
//offset = 0
//limit = 10

graph_of_word_query: {
  query = g.V().has("name", within(queryTokens))

  if (query.clone().count().next() < 1) return [[results: [:], numDocs: 0]]

  indegreePerTokenPerDoc = query.clone()
    .project("v", "indegree").by()
    .by(inE().values("doc_id").groupCount())

  docFrequencyPerToken = query.clone()
    .project("v", "docFreq").by()
    .by(bothE().groupCount().by("doc_id"))
    .collectEntries { e -> [(e["v"]): e["docFreq"].size()] }

  docLengthsPipe = g.E().group().by("doc_id").by(inV().count())

  docLengths = []

  docLengthsPipe.clone().fill(docLengths)

  if (docLengths.isEmpty()) return [[results: [:], numDocs: 0]]
  avgDocLength = docLengthsPipe.clone()[0].values().sum() / docLengthsPipe.clone()[0].values().
      size()

  corpusSize = g.E().values("doc_id").unique().size()

  twIdf = indegreePerTokenPerDoc.clone().collect { token ->
      token['indegree'].collect { docID, indegree ->
        score = twIdf(indegree, docFrequencyPerToken[token['v']], docLengths[docID][0],
            avgDocLength, corpusSize)

        [
         docID: docID,
         twIdf: score,
         components: [
           docID: docID,
           'tw(t, d)': indegree,
           b: 0.003d,
           '|d|': docLengths[docID][0],
           avdl: avgDocLength.doubleValue(),
           N: corpusSize,
           'df(t)': docFrequencyPerToken[token['v']],
           'tw-idf(t, d)': score
         ]
       ]
      }
    }
    .flatten()
    .groupBy { item -> item['docID'] }
    .collect { docID, item -> [docID: docID, score: item['twIdf'].sum(), components: item['
        components']] }
    .sort { -it.score }

  numDocs = twIdf.size()

  twIdf = twIdf
    .drop(offset)
    .take(limit)

  [[results: twIdf, numDocs: numDocs]]
}
```

Graph of Entity

```
//queryTokens = ['born', 'new', 'york']
//queryTokens = ['musician', 'architect']
//offset = 0
//limit = 5

graph_of_entity_query: {
  query = g.withSack(0f).V().has("name", within(queryTokens))

  if (query.clone().count().next() < 1) return [[results: [:], numDocs: 0]]

  termEntityFrequency = g.V().outE("before")
    .dedup()
    .where(inV().has("name", within(queryTokens)))
    .project("entity", "term")
    .by { p = g.V().has("url", it.value("doc_id")); if (p.hasNext()) return p.next() else
        return none }
    .by(inV())
  .where(__.not(select("entity").is(none)))
  .group()
    .by(select("entity"))
    .by(count())
  .unfold()
  .collectEntries { [(it.key): (1 + Math.log(it.value))] }

seedScoresPipe = query.clone()
  .union(
    __.out("contained_in"),
    __.where(__.not(out("contained_in"))))
  .choose(
    has("type", "entity"),
    group()
      .by()
      .by(
        fold()
          .sack(sum).by(count(local))
          .sack(div).by(unfold().in("contained_in").dedup().count())
          .sack()
      ),
    group()
      .by()
      .by(constant(1d))
  )
  .unfold()
  //.order()
  //.by(values, decr)

seedScores = seedScoresPipe.clone()
  .collectEntries { [(it.key): (it.value)] }

maxDistance = 1

// shortest path
distancesToSeedsPerEntity = seedScoresPipe.clone()
  .select(keys).as("seed")
  .repeat(both().simplePath().where(neq("seed")))
  .until(
    has("type", "entity")
    .and()
    //.outE().where(__.not(hasLabel("contained_in"))) // retrieval unit
    .has("doc_id")
    .or()
    .loops().is(eq(maxDistance))
  )
  .where(
    __.has("type", "entity")
    .and()
    //.outE().where(__.not(hasLabel("contained_in"))) // retrieval unit
    .has("doc_id")
  )
  .path().as("path")
  .project("entity", "seed", "distance")
    .by { it.getAt(it.size() - 1) }
    .by { it.getAt(2) }
    .by(sack(assign).by(count(local)).sack(sum).by(constant(-2)).sack())
  .group()
```

```
      .by(select("entity"))
      .by(group().by(select("seed")).by(select("distance").min()))
    .unfold()

  ewTef = distancesToSeedsPerEntity.clone().collect {
      docID = it.key.value("doc_id")
      coverage = it.value.size() / seedScores.size()

      // Iterate over each seed.
      weight = it.value.collect { s ->
        seedScores.get(s.key, 0f) * 2f / (1f + s.value)
      }
      avgWeightedInversePathLength = weight.sum() / weight.size()
      entityWeight = coverage * avgWeightedInversePathLength

      [
        docID: docID.toString(),
        score: entityWeight,
        components: [[
          docID: docID.toString(),
          'c(e, S)': coverage.doubleValue(),
          'w(e)': avgWeightedInversePathLength,
          'ew(e, E, b)': entityWeight.doubleValue(),
          'tef(t, e)': termEntityFrequency.get(it.key, 0),
          'ew-tef(q, e)': score
        ]]
      ]
    }
    .plus(termEntityFrequency.collect {
      docID = it.key.value("doc_id")

      [
        docID: docID.toString(),
        score: 0d,
        components: [[
          docID: docID.toString(),
          'c(e, S)': 0d,
          'w(e)': 0d,
          'ew(e, E, b)': 0d,
          'tef(t, e)': it.value.doubleValue(),
          'ew-tef(q, e)': score
        ]]
      ]
    })
    .unique { a, b -> a.docID <=> b.docID }
    .sort { -it.score }

  numDocs = ewTef.size()

  ewTef = ewTef
    .drop(offset)
    .take(limit)

  [[results: ewTef, numDocs: numDocs]]
}
```