Fast Search for Named Entity Recognition

Summer 2012: Basic Algorithms for Computational Linguistics

David Przybilla[*], Sibel Ciddi[†], Noushin Fadaie[‡], Moinuddin Haque[§]

Universität des Saarlandes
Computational Linguistics & Phonetics,
Saarbrücken, Germany

August 17, 2012

Abstract

In this paper, we present three different approaches based on Approximate Dictionary Matching Algorithm that uses similarity measures for the retrieval of Named Entities. For the evaluation, we extracted PERSON Named Entity types from the CoNLL-2003 Shared Task Corpus Data. In order to determine the fastest search approach, for the results, we compare CoNLL data the average processing times outputted by these different dictionary implementations.
keywords: Information Extraction, dictionary search, similarity measures, simString, Named Entities, NEs, NER.

1 Introduction

Due to the problematic identification of Named Entities, collecting Named Entities in the field of Information Extraction and Natural Language Processing has been a popular task during the past years. There have been more than several methods that deal with this specific task; both from a statistical point of view, and rule-based method point of view, for extraction of Name Entities(NE) from raw text. On this paper, we present `NERSimString`, which is a Java library, based on an approximate dictionary matching algorithm[1][1]that uses various similarity measures. This algorithm targets the retrieval of strings faster for Named Entity Recognition systems. The main idea behind the `NERSimString` is that it creates a library for storing Name Entity dictionaries, and at the same time provides a quick Named Entity annotation tool for the given named entity dictionaries generated by the system.

[*]dav.alejandro@gmail.com
[†]sciddi@coli.uni-sb.de
[‡]nfadaei@coli.uni-sb.de
[§]moinuddinmushirul@gmail.com
[1]http://www.chokkan.org/software/simstring

In most NER systems, creating and storing NE dictionaries can raise some problems. Even in most state-of-the-art NER systems, generated NE dictionaries, to be used as fixed-lists later, impose the following issues that still remain challenging to resolve:

- Even in a restricted domain, the number of Named Entities in a dictionary can be overwhelming.
- Considering the size of a corpus, when the fixed-dictionaries are given, the final goal of finding all the NEs contained in a corpus can still be a hard task.
- Even in semi-supervised methods that are trained to recognize the NEs require fast searching tools; because the training requires searching and finding the location of a given set of NE that is located in a corpus.

Therefore, in order to make the annotation task of Named Entities a tractable problem, a method that looks for words quickly in a huge dictionary becomes mandatory. The `NERSimString` algorithm precisely targets to handle this specific problem and aims to achieve an accurate and fast searching method.

2 The SimString Method

For more straightforward and simpler NER tasks with a narrower focus, exact dictionary look-up methods can be used as well. However, besides non-user generated data, the SimString method targets the user generated data *(i.e., social network corpus)* as well. Because user-generated data can have many variations, including potentially misspelled words, while dealing with user-generated data, `NERSimString` offers fast, approximate string retrieval. In these terms, it provides a quick way to make the search on the cost of space.

In the `NERSimString` algorithm, `NerSimString` dictionary is created from a list of items made of one Named Entity–*which can be multi-word*–per line. Fast search queries can be requested via this dictionary. On this paper, for the purpose of comparing different dictionary set-ups; we adopted three different low-level implementations[2]:

1. **SuffixTree Implementation:** A dictionary based on a suffix tree to hold in memory
2. **Naive-HashTable Implementation:** A dictionary based on hash tables to hold in memory
3. **MemoryMapped Hashtable Implementation:** A dictionary based on hash maps, that is meant for large amounts of data. In order to avoid being completely loaded into memory, it is dynamically stored in the disk.

In order to provide approximate string retrieval, the following similarity measures have been used:

[2]The implementation source code is available at: https://github.com/dav009/NERSimString

i. Jaccard

ii. Dice

iii. Cosine

With these settings, `NERSimString` works fast by creating an inverted list of the size of n-grams. Based on one of the given similarity measures and a specific threshold, it measures how many n-grams at a certain length are the most likely to match the given NE query. This enables the system to discard as many alternatives as possible, which then makes the fast search possible. Thus, given these settings, the SimString model can be used both for the annotation of Named Entities and for the extension of current NER systems that are trained to predict Named Entities.

3 The SimString Algorithm

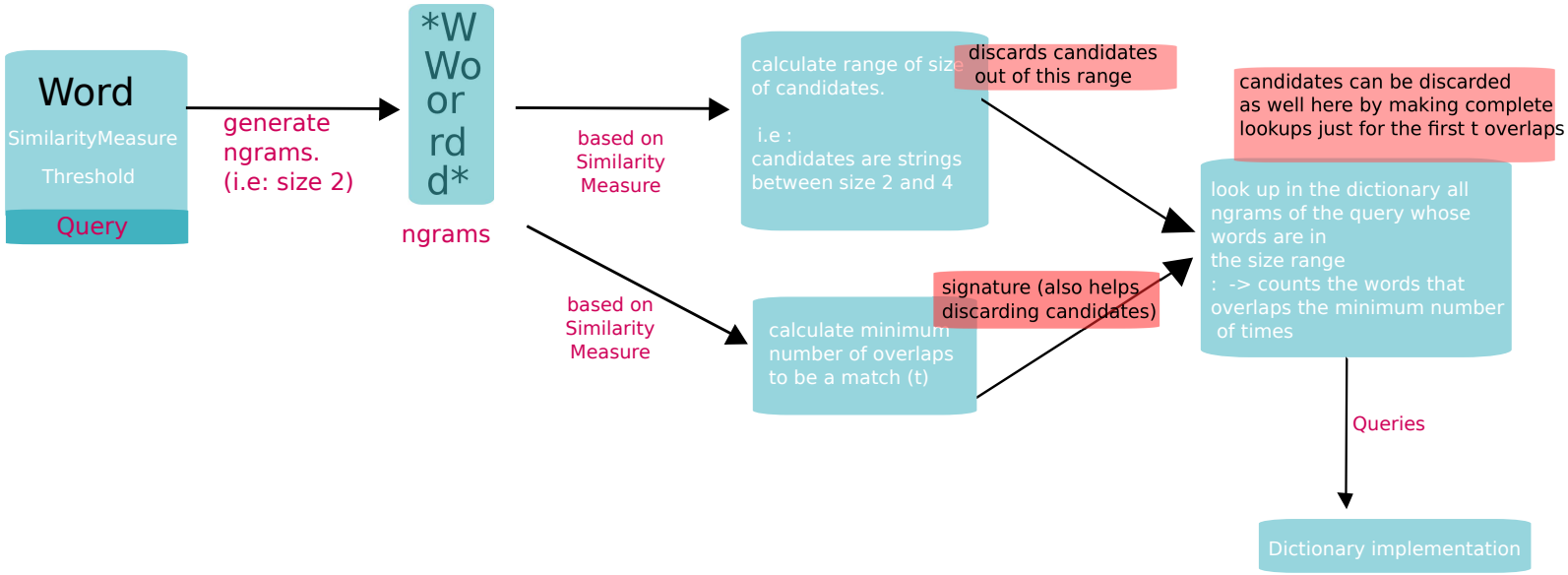


Figure 1: Overview of the SimString Algorithm

4 Implementation

In order to allow further experimentation and expansion of the tool, we aimed to implement a model with a highly object oriented framework so that the different parts of the tool could be replaced by new ones smoothly. Therefore, for the writing of the script, Java programming was used for two reasons: First of all, Java is a popular programming language; and second, it opens up the possibility of using the tool with many other packages or even as a webservice for more sophisticated applications.

4.1 Java Packages

Description of the most important packages and classes related to the implementation:

IO

- The classes in this package deals with the input & output files.
- The class `DictionaryReader.java` is in charge of reading a raw dictionary file *(one entry per line).*

Measures

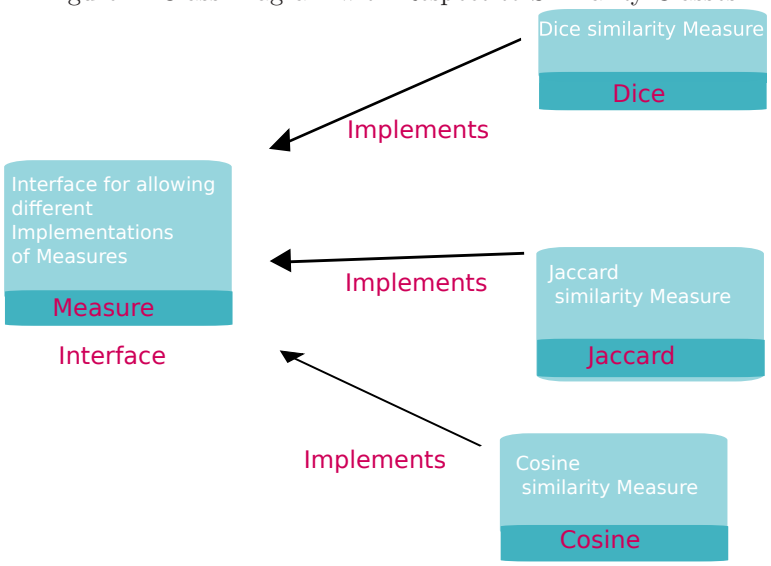
- This package consists of all the classes related to the different similarity measures.
- The SimString algorithm works independently from the similarity measure that is choosen initially.
- A separate interface, named `Similarity` that chooses the similarity measure exists within the package.
- For the extension of the system to provide new similarity measures, this interface has to be implemented.

Additionally, `Measures` package includes another class, named `"MeasureFactory"`. This class is in charge of constructing the similarities' of objects.

Dictionary

- This package contains all the classes that are responsible for dictionaries.
- Besides the dictionary classes, this package also contains other classes related to the different implementations of dictionaries.
- One of the most important classes, named `LowLevelDictionaryImplementation`, is an interface that allows the system to implement different kinds of dictionaries.
- With this interface, a dictionary can be represented either as a hash table or as a suffix tree.
- Despite these dictionaries' different data structures, the `LowLevelDictionaryImplementation` interface makes them transparent to the SimString algorithm.

Figure 2: Class Diagram with Respect to Similarity Classes



SimString

- This package consists of the class `SimString`, which is responsible for retrieving all the similar Named Entities from the dictionaries.
- With this class, for each dictionary, a similarity configuration and a query can search the dictionary and retrieve all the similar Named Entities according to its parameter values.

Util

- This package consists of several classes with different responsibilities.
- The `N-gram` class is in charge of handling specific n-gram functionalities, such as splitting words into n-grams.

Examples

- This package contains classes –*each one of which is an example*– for the potential users.

Test

- This package contains classes that allow developers to asses the time performance of the tool.
- For evaluation purposes, it makes comparisons of the different implementation models.

4.2 Dictionary Implementations

As mentioned in the previous sections, as an addition to the `SimString` algorithm, we introduced three different models to implement the underlying data structure that represents the n-gram-inverted index. For the representation of this, we used two different data structure types:

i. `Mapped Data Structure:` *In this data structure type, data structures are saved in the hard disk, and small chunks of the data structures are loaded into memory on an on-demand / as-needed base.*

ii. `Not-Mapped Data Structure:` *As it is the case with the traditional data structures, in this data structure type, the data structures are kept entirely on memory while the program is running.*

The memory mapped data structures are ideally used for cases when the number of data that is being stored is very high, and thus having the whole data structure in memory at once is not always feasible. However, the memory mapped data structures come with a cost–that is–`Memory`.

Because the memory mapped data structure implementations aim to be as fast as traditional data structures; they have to be stored in memory in such a way that they can be retrieved quickly. The only way that enables this is achieved by having a fixed number of bits used per data field in the mapped data structure. In other words, this means that no matter what the size of the data is, a slot in this structure will always have a fixed-size. However, having this specific structure with a fixed-size slot results in using extra memory.

4.2.1 SuffixTree

This is an implementation of the inverted index of n-grams, using suffix trees. The keys in this model are made from the strings of the shape `'ngram-sizeOfString.'` The values are priority queues with the IDs of dictionary entries, in which the size is `sizeOfString`, consisting of the given `ngram`.

4.2.2 Naive-HashTable

This is an implementation of the inverted index of n-grams that uses Java HashMaps. The keys in this model are made from the strings of the shape `'ngram-sizeOfString.'` The values are priority queues with the IDs of dictionary entries, in which the size is `sizeOfString` and contains the given `ngram`.

4.2.3 MemoryMapped Hashtable

This is an implementation of the inverted index of n-grams that uses memory mapped hash tables. It enables the generated dictionary to be saved in a file, and to be loaded for later use.

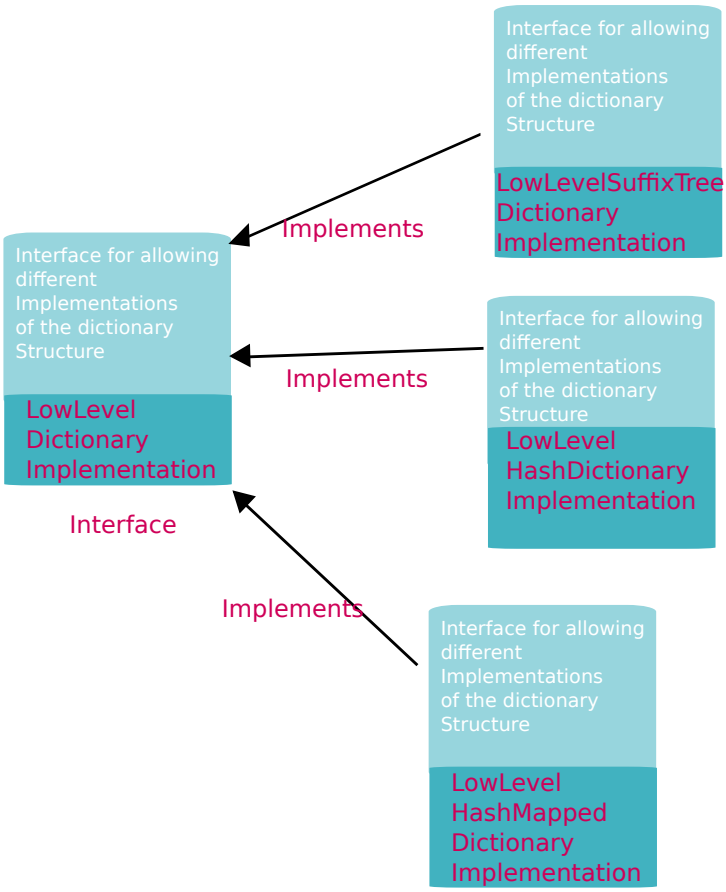


Figure 3: The Dictionary Implementations

5 Evaluation and Results

`NERSimString` algorithm with three different dictionary implementations were evaluated by using the CoNLL-2003 Shared Task Data Corpus[3] [4]. We compared the dictionary implementation models against the original SimString algorithm implementation as the baseline. The evaluation of these different implementations measured the average processing time/*per second* of the dictionaries by considering the given `n-gram` sizes and different similarity `thresholds`.

We evaluated these dictionaries on two different CoNLL data settings, only for PER (*person_NE*) Named Entity type:

i. First, we tested the dictionaries by taking 200 **random** search strings per method.

ii. Second, we tested the dictionaries by taking the first 200 words from the CoNLL corpus.

[3]Data files available at: http://www.clips.ua.ac.be/conll2003/ner/
[4]English data set request from: http://trec.nist.gov/data/reuters/reuters.html

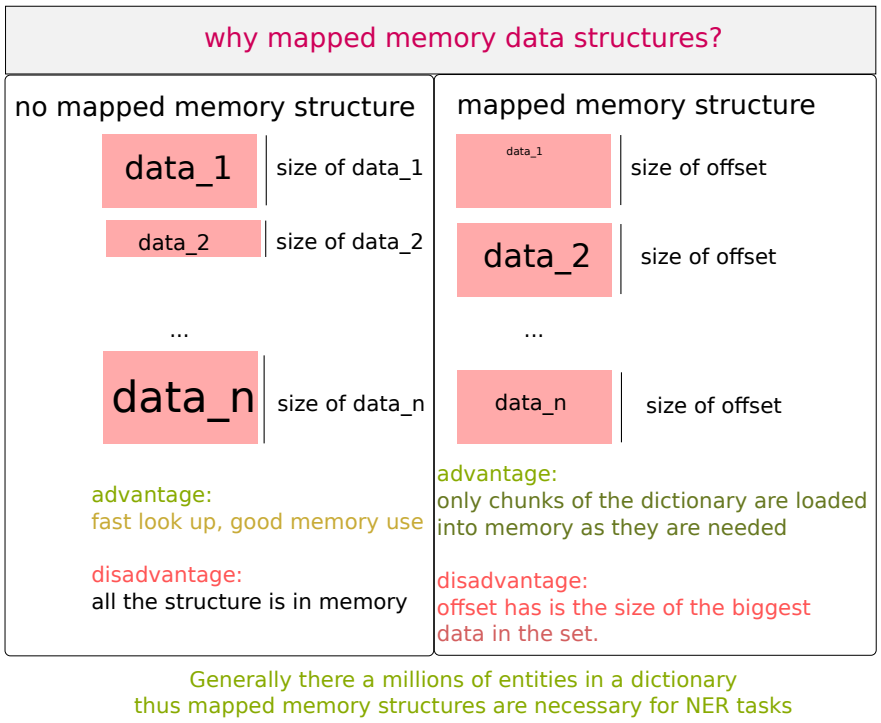


Figure 4: Comparison between Mapped & Not-Mapped Data Structures

The following tables show the results for the average processing times returned by each of the dictionary implementations:

	SuffixTree	**HashTable**	**Mapped HashTable**	**SimString**
`Threshold`	0.2	0.2	0.2	0.2
`N-Gram:3-Avg.Time`	0.26034846156	0.1600577245	0.38093002297	0.1198204536
`N-Gram:5-Avg.Time`	0.13545810821	0.1907321242	0.13516753774	-
`N-Gram:8-Avg.Time`	0.18630211667	0.1774560892	0.19634142339	-

Table 1: Average Look-up Time Results Based on 200 Random Strings

	SuffixTree	**HashTable**	**Mapped HashTable**	**SimString**
`Threshold`	0.8	0.8	0.8	0.8
`N-Gram:3-Avg.Time`	0.317907188	0.611821308	1.132227919	2.774974599
`N-Gram:5-Avg.Time`	0.448093155	0.599589026	1.480562861	-
`N-Gram:8-Avg.Time`	1.003509663	2.200090178	1.927359551	-

Table 2: Average Look-up Time Results Based on First 200 Strings on CoNLL Data

These results show that the `SuffixTree` and the regular *(naive)* `HashTable` Dictionary implementation outputs were relatively close to the original `SimString` algorithm implementation. The `Mapped HashTable` Dictionary handled the data input efficiently as well, however due to the construction of memory mapped structure, the average look-up times resulted in more time than the other two dictionary implementations.

6 Conclusion and Future Work

On this paper, we presented different implementations of the proposed `SimString` algorithm. We found out that the implementation of `SimString` is easily extensible for different approaches. Some of the implementations are more useful for other tasks like fast look-up, and for Named Entity Recognition, which is one of the problematic areas in natural language processing and information extraction. We found out that the `SuffixTree` and `HashTable` dictionaries returned the relatively similar and fast average look-up times for potential Named Entity candidates. With the different implementations of the inverted dictionary indexes we have presented, we can conclude that they provide different applications, and they were useful in evaluating the performance of the implementation for comparison purposes.

Some of the research areas that can be looked further in detail, for example, would be building a `bootstrapper` for learning Named Entities based on the different implementations of the `SimString` algorithm, such as the `NERSimString` dictionaries we presented on this paper. Adding extra similarity measures alternatives for testing and training might also help finding out the best approach for that kind of applications.

More sophisticated NER systems can benefit from the `SimString` implementations by further expanding it with *signature information* which can be extracted from the dictionaries. This info could include anything from finding the *upper and lower boundaries* of NE candidates to applying Noun-Phrase Chunking on the NE Candidates returned from the dictionaries.

Besides the potential *bootstrapping* oriented implementation or more sophisticated NER system; adding more support for `Memory-Mapped HashTable` dictionaries might also be useful for other applications. For the basic NER system, it is expected to see that the `Mapped HashTable` Dictionaries outputted the slowest search look-up times for Named Entity candidates. However, this does not mean that the `Mapped HashTable` dictionaries are not a good candidate implementation for other look-up applications, that require a lot of data to be held on the memory. For example, for spell-checker application dictionaries, especially for languages with a large lexicon *(e.g., highly-inflected languages)*, the use of `Mapped HashTable` dictionary implementation would be very beneficial.

As a final note, for this project, we would like note that researching about different implementation techniques and learning about the differences in application to search look-up was an enjoyable learning task. Because information extracting and search techniques is one of the most challenging–and at the same

time rewarding tasks in NLP, we plan on continuing on working on the mentioned research areas for future research.

References

[1] Naoaki Okazaki and Jun'ichi Tsujii. Simple and efficient algorithm for approximate dictionary matching. In *Proceedings of the 23rd International Conference on Computational Linguistics (Coling 2010)*, pages 851–859, Beijing, China, August 2010.