Fast Search for NER
Basic Algorithms for Computational Linguistics Project

Sibel Ciddi[*] Noushin Fadaie[†] David Przybilla[‡]

July 14, 2012

1 Introduction

NERSimString is a Java library based on SimString [1] paper for doing fast approximate String retrieval.

The main idea of*NERSimString* is to provide a library for creating Name Entitiy dictionaries and a fast NE annotation tool given the generated entities dictionaries.

Gathering Name entities has been a popular tasks during the past years, there are both statistical methods and rule based methods which are good at the task of extracting Name Entities(NE) from raw text.
By Using this tools long list of NE can be created and stored in dictionaries, so that later can be used as a fixed list.

However the following problems arise:

- Even in a restricted domain, the number of NE in a dictionary can be overwhelming.
- Given fixed dictionaries the final goal of finding all the entities contained in a corpus can be a hard task given the size of the corpus.
- Even semi-supervised methods for learning to recognize NE require fast search tools, since learning requires for example to find where a given set of NE are located in a corpus.

So a way for looking for words quickly in a huge dictionary is necessary in order to make the annotation task of NE a tractable problem, this is the aim of NERSimString.

[*]sciddi@coli.uni-saarland.de
[†]fadaei.noushin@gmail.com
[‡]dav.alejandro@gmail.com

Why not exact lookups?

Exact lookups are ok for some tasks, how ever not for NE recognition and specially when dealing with user-generated data (i.e: social networks) why? because there can be variations or miss spelled words, that's why $NERSImString$ offers fast-Approxiamte String Retireval

What does NERSimString offer?

It offers a quick way to make search on the cost of space.

Basically $NERSimString$ receives a dictionary with a NE entry per line(this entry can be multiword) as a result it creates a $NerSimString$ dictionary in which fast queries can be done. Those dictionaries come in different flavors, according to different low-level-implementation

- **SuffixTree:** it is a dictionary based on a suffixTree hold in memory
- **Naive-HashTable:** it is a dictionary based on hashTabes hold in memory
- **MemoryMapped Hashtable:**it is a dictionary based on hashmaps,it is meant to be for huge amoutns of data since it is stored dinamically in the disk in order to avoid being loaded completely into memory.

What kind of similarities?

In order to provide approximate String retrieval the following similarity measures are supported:

- Jaccard
- Dice
- Cosine

As a result this tool can be used either to annotate or to extend current systems that learn to predict NE.

Why is it fast?

$NERSimString$ works creating an inverted list of ngrams-size. Basically given a similarity measure, and a threshold it measures how many ngrams of which size should for sure match the given query in order to accomplish the minimum given threshold. by doing this many possible alternatives are discarded and thus fast search is possible.

Where is the code available?

the repository is hosted at github : https://github.com/dav009/NERSimString

The Benchmarking

We

Future work relating this can involved:

* adding support for more memoryMapped structures * building a bootstrapper (for learning NE) based on NERSimString * adding extra similarity Measures alternatives

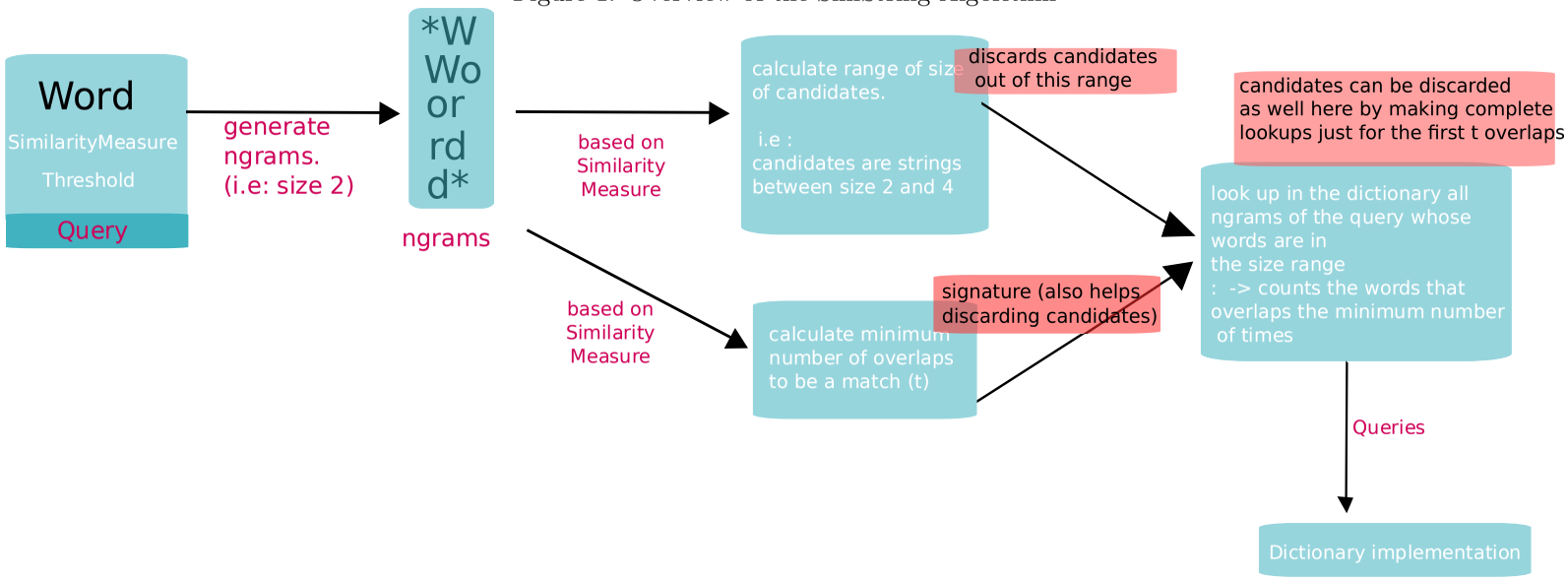
*

Sections of possible document: Introduction Problem Description Solution (Description of SimString algorithm) Implementation Results Conclusions

[1] http://www.chokkan.org/software/simstring/

2 SimString Algorithm

Figure 1: Overview of the SimString Algorithm



3 Implementation

The concept behind the implementation was to make a highly object oriented framework so that the different parts could be replaced by new ones transparently and thus allowing for further experimentation and expansion of the tool. The chosen language for the implementation was Java, since Java is a popular programming language it opens up the possibility of using the tool with many other packages or even as a webservice for more sophisticated applications.

3.1 Packages

This is a description of the most important packages and classes related to the implementation:

IO

The classes in this package have the responsability of dealing with the input/output files.
The class DictionaryReader.java is incharged of reading a raw dictionary file (one entry per line).

Measures

This package is composed of all classes relating the different similarity measures. The SimString algorithm is independant of the chosen similarity, for doing this an interface called *Similarity* exists within the package, in order to extend the system to provide new similarities measures this interface has to be implemented.

Aditionally this package has a class name MeasureFactory, which is incharged of constructing similarities's objects.

Dictionary

This package contains all the classes which have a responsability related to dictionaries.

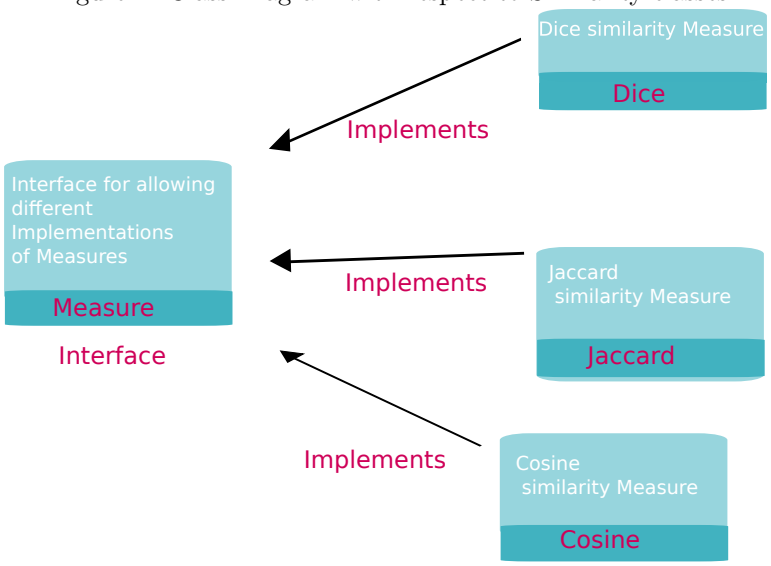
Among the most important ones are *LowLevelDictionaryImplementation* this is an interface which allows the tool to implement differnt kinds of Dictionary Implementations, for example one dictionary can be represented either as a hashtable or as suffixtree, while those data structures are different by implementing *LowLevelDictionaryImplementation* they become transparent to the simString algorithm.

Additionally this package contains all the classes related to the different offered implementations of dictionaries.

SimString

This package contains the class *SimString* which given a dictionary,a similarity configuration and a query is able to search in the dictionary and retrieve all the

Figure 2: Class Diagram with respect to Similarity classes



similar NE given the value of the parameters.

Util

This package is composed of different classes with different responsabilities. The *NGram* class is incharged of splitting words into ngrams and dealing with any specific functionality related to ngrams.

Examples

This package contains classes (each one is an example) for the potential users.

Test

This package contains classes which allow team to asses the time performance of the tool and make comparisons among the different implementations.

3.2 Dictionary Implementations

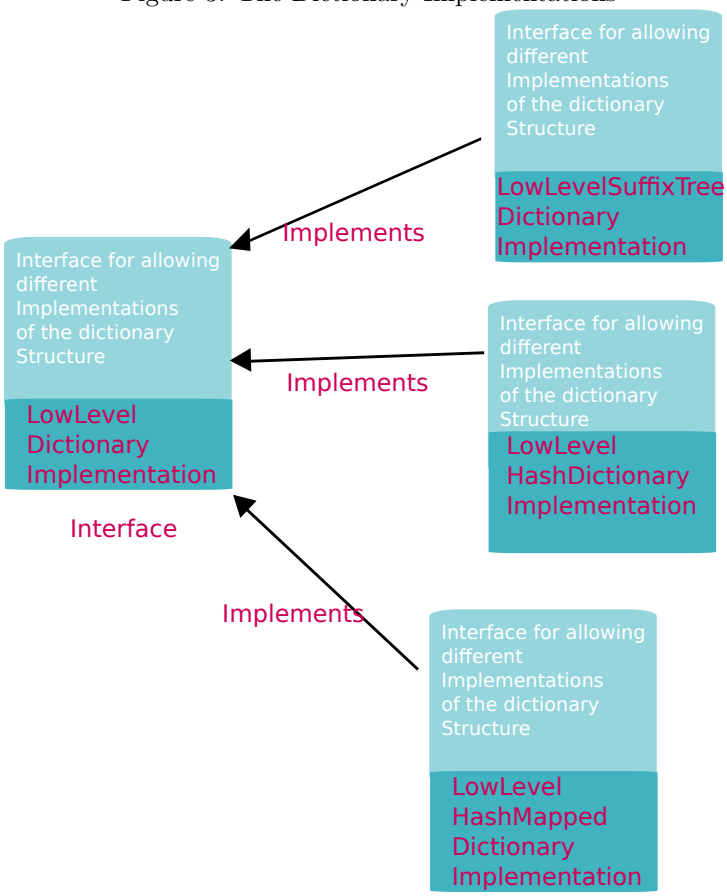
As an addition to the simstring algorithm, the team introduced different ways to implement the underlying data structure representing the ngram-inverted index.

There are two types of data structures alternatives: mapped and not mapped. Not mapped data structures are traditional data structures, that is, structures that are kept entirely on memory while the program is running.
As opposed to this type there are mapped data structures which are data structures saved in the hard disk and as data from it is needed small chunks of the datastructure are loaded into memory.

The memory mapped data structures are ideally for cases when the number

of data being stored is very high and having the whole datastructure in memory at once is not feasible. However memory mapped data structures come with a cost: $Memory$.
Since memory mapped data structure implementations want to be as fast as traditional data structures, they have to be stored in memory in a way they can be queried quickly.
The only way to assure this is by having a fixed number of bits used per data field in the mapped data structure,this means, that no matter the size of the data, a slot in this structures will always have a fix size, this means using extra memory.

Figure 3: The Dictionary Implementations



3.2.1 SuffixTree

This is an implementation of the inverted index of ngrams by using suffixTrees. The keys in this case are strings of the shape: $ngram\text{-}sizeOfString$, the values are Priority Queues with the id's of dictionary entries which size is $sizeOfString$ and contains the given $ngram$.

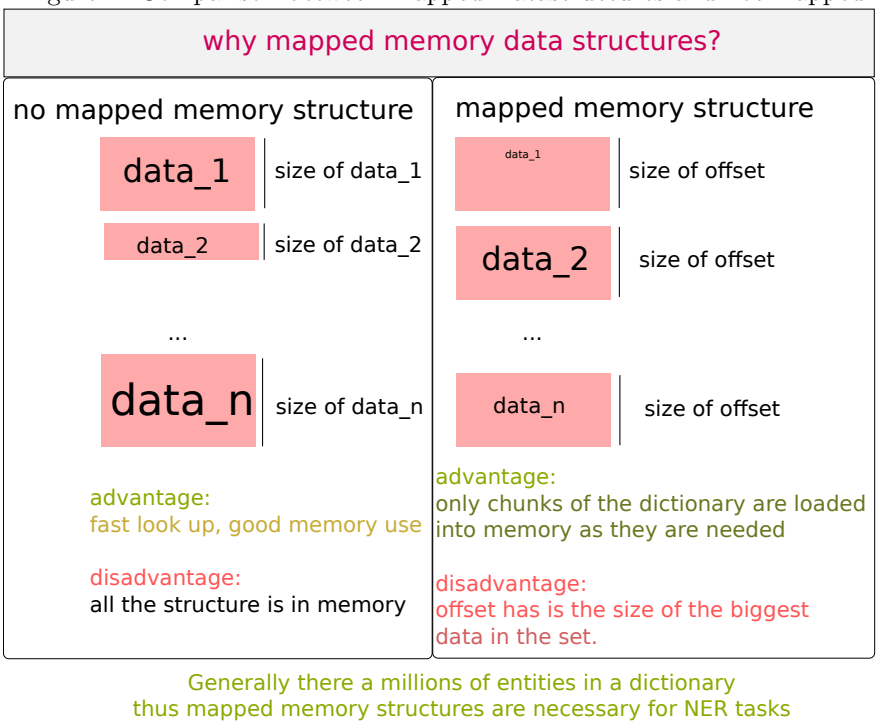
3.2.2 Naive-HashTable

This is an implementation of the inverted index of ngrams by using Java HashMaps. The keys in this case are strings of the shape: $ngram\text{-}sizeOfString$, the values are Priority Queues with the id's of dictionary entries which size is $sizeOfString$ and contains the given $ngram$.

3.2.3 MemoryMapped Hashtable

This is an implementation of the inverted index of ngrams by using memory mapped Hashtables.
It allows to save the generated dictionary in a file and to load it for later use.

Figure 4: Comparison between Mapped Datastructures and not mapped



4 Testing and Results

5 Conclusion and Future Work