UNIVERSITY OF PIRAEUS DEPARTMENT OF DIGITAL SYSTEMS

Link Discovery on the Web of Data

Time and Space Efficient Large Scale Link Discovery using String Similarities

2/11/2017

Postgraduate student: Andreas Karampelas

Supervisor: Professor George A. Vouros

This work proposes and evaluates a time and space efficient approach for computing links between a source data set and a target dataset by exploiting string similarities. Evaluation results show the time and space efficiency of the proposed method against state-of-the-art approaches for link discovery.

Acknowledgements

I wish to express my gratitude to my thesis supervisor, Professor George A. Vouros for his tireless help and guidance, but also for his encouragement and valuable advice throughout my research for this work.

I would also like to thank Post Doctorate student George Santipantakis for his generous assistance for solving many practical problems that occurred.

Table of Contents

List of Figures

List of Algorithms

List of Tables

List of Graphs

1. Abstract

This work proposes and evaluates a time and space efficient approach for computing links between a source data set and a target dataset by exploiting string similarities among entities' properties. The proposed approach builds on a basic indexing method that facilitates pruning dissimilar pairs and supports effective verification of candidate pairs. It proposes a blocking method that organizes the target data set appropriately, to perform queries concerning matching a specific string. It supports an effective filtering approach that uses three filters that lead to a relatively small amount of candidate strings that need verification. Lastly, for the verification of each candidate string it uses an optimized algorithm for computing the edit distance between two strings. Evaluation results show the time and space efficiency of the proposed method against state-of-the-art approaches for link discoveries.

2. Introduction

The web of data forms a data space where data sources are connected via RDF links. The (semi-)automatic computation of such links is a problem that is related to the time and space efficiency of the computations, to the precision and recall of the computations, and of course to the involvement of human guiding (e.g. via link specifications) and verifying the computations: Overall, the challenge of computing these links for large data sets is big. Therefore, the need for scalable, automatic and reliable (in terms of precision and recall) link discovery tools that scale to very large data sources is growing fast, as the web of data is growing.

Using the widely used filter-verification framework, proposed methods aim to (a) apply filters early enough in the process to prune large numbers of dissimilar pairs and generate a small number of matching candidates; and (b) reduce the complexity of verifying each candidate pair towards computing similarities and outputting the final results.

To dismiss non-matching pairs prior to verification (i.e. to prune the candidate pairs) there are many generic methods, the most-well known of which are those mentioned as blocking methods [18]. As it is known, standard blocking techniques lead to decrease of recall due to false dismissals [3]. While blocking techniques create blocks in one dimension, the multiblocking technique used in SILK [26] creates multiple blocks for a number of properties in multiple dimensions. This increases the time efficiency of the link discovery process significantly, in high precision and without sacrificing recall.

Regarding the computation of links using string similarities, there are many filtering techniques that have been proposed [25].

Also, while the emphasis is on filtering (which is reasonable given that the need to effectively prune the search space in scalable ways is big), the need to further improve the verification stage emerges. This is true especially in cases where strings are large enough. Many state-of-the-art algorithms for reducing the complexity of computations for comparing (large) strings do exist [9].

This work proposes and evaluates a time and space efficient approach for computing links between data sources, exploiting string similarities. The proposed approach builds on a basic indexing method that facilitates pruning dissimilar pairs and verifying candidate pairs, effectively. It proposes a filtering approach that utilizes string lengths and further partitions the target datasets to blocks of entities according to dissimilarities between strings of equal length. For the verification stage, this work presents and utilizes methods that aim to reduce the complexity for computing the string metric used to quantify the similarity between two strings.

Motivation

Linked Data refers to data published on the Web in such a way that the machine is able to read these data, their meaning is defined explicitly, they are linked to external links and they can be linked from external links[20].

Tim Berners-Lee [11] created a set of rules, so that the data published can be interconnected. These rules are the following:

1. Use URIs as names for things

2. Use HTTP URIs so that people can look up those names

3. When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL)

4. Include links to other URIs, so that they can discover more things

The RDF Vocabulary Definition Language (RDFS) [20] and the Web Ontology Language (OWL) [17] are used to create vocabularies that describe entities that exist on the Web as well as their relation to other entities.

Over the last years there is an increasing interest on Link Discovery frameworks, while the web of data is growing. This interest is analog to the growth of the amount of data which is published as Linked Data [24]. So, there is undeniable and increasing demand for efficient and reliable tools for discovering and maintaining data links between data sources on the Web of Data. The main task that these tools have to perform is to identify semantically similar instances in different data-sources and link them.

3. Related work

a. Related problems

A common statement of the Link Discovery (LD) problem is the following: Given two sets of instances S, T find the matching pairs in S x T, that would be related with the OWL property **owl:sameAs** [17]. The Link Discovery problem is closely related with the problem of Record Linkage, which is the process of detecting pairs of matching instances between individually clean(duplicate free) but overlapping sets of instances. This problem is systematically examined in the following technical reports [29, 30]. There are several other names used from researchers for this problem like object identification, data cleaning, approximate matching (joins) [1]. Also it is related to the task of De-duplication, which has a set of instances S as an input and detects the duplicate instances that this set S contains [12]. Also these two tasks are often referred as the Entity Resolution (ER) problem [18]. Although similar techniques are used to address the ER and LD problems, different tools and software has been developed because they still have significant differences [12].

The rationale behind the software used to address these problems is to reduce it to a problem of computing the similarity of instances, based on properties of these instances. These properties are expressed or represented by common data-types such as arithmetical values, characters or strings. The similarity metric used to perform the matching task depends on the data-type of these properties.

In the case that a property ranges to strings, then these frameworks utilize a string similarity metric. A very common string similarity measure used is the Levenshtein distance [8], but there are many other string similarity metrics such as Jaro – Wrinkler metrics [30], which are primarily intended for short strings. There is an extended literature on string similarity metrics and many experimental comparisons between them [23]. We will examine thoroughly string similarity algorithms in the next chapter.

Another related problem to Link Discovery, when considering properties ranging to string values, is called “String Similarity Joins” [7]. The formulation of the problem is the following: *Given two sets of strings R, S and a similarity metric, the string similarity join problem is to find the set { (r, s) | r ∈ R, s ∈ S, r and s are similar w.r.t. the similarity metric}*. There are many algorithms proposed to solve this problem. Some of them use an optimized version of the prefix filtering technique, which is the most effective filtering technique to address to this problem. The following 7 algorithms are proposed to optimize the prefix filtering concept: AllPair [2], PPJoin [32], EDJoin [31], TrieJoin [5], QChunk [19], VChunk [28], AdaptJoin [27]. Others use different signature schemes than prefix filtering like PartEnum [1], PassJoin [9], FastSS [4].

A comprehensive survey on the existing string similarity join algorithms can be found in this work “String Similarity Joins: An Experimental Evaluation” [7]. The main contribution of this work is that it offers a thorough experimental comparison of string similarity algorithms under the same experimental framework utilizing a variety of authentic datasets. According to the findings of this work the PassJoin algorithm outperforms other algorithms in terms of both running time and memory usage. Also the work that presented the PassJoin algorithm [9] showed that this algorithm achieved better results than EDJoin and PPJoin.

b. Proposed methods – Algorithms - Frameworks

Several frameworks have been proposed for link discovery (LD) between instances in different knowledge bases, such as SILK[22] and LIMES[10]. An overview of these frameworks is available in the paper "A Survey of Current Link Discovery Frameworks" [12]. The main contribution of this work is that it offers a generic schema of the link discovery frameworks and a comprehensive evaluation of the performance and characteristics of current link discovery frameworks.

Most state-of-the-art LD frameworks have a similar workflow which consists of a preprocessing module, a matching module and a post-processing module. The preprocessing module deals with tasks such as data cleaning or finalizing the linking specification. The post-processing module usually deals with tasks such as determining the type of mapping needed or applying a matching rule. For example we could restrict the matching task so that each instance is linked with at most one instance and this instance is the optimal fit choice. The matching module is the heart of the framework and it consists of three main steps: The first step is the *blocking* step, where a property or an attribute is being used as a blocking key to divide the target dataset to partitions. The output of this step usually creates the appropriate data structure to implement the next two steps. The next step is the *filtering* step, where several criteria and methods are being used to prune as much as possible the number of instances that need verification. The output of this step is a relatively small set of candidate instances that have to be verified. The last step is the *verification* step, where this set of candidate instances is checked to see if every instance meets the specified criteria.

The following schema shows the general workflow of the LD frameworks where you can see the main modules used as well as how they are related:

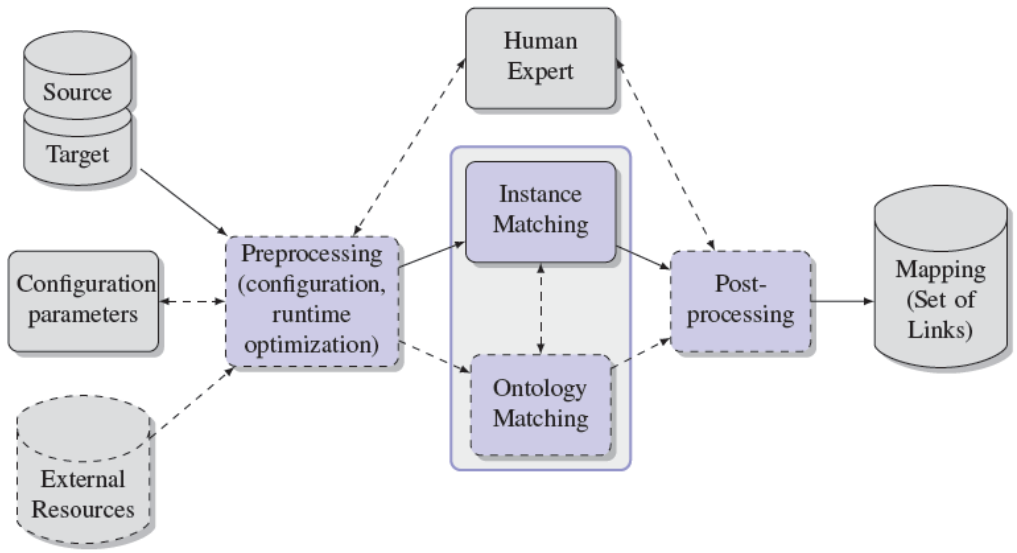


Fig. 1. General workflow LD Frameworks (steps with dashed borders are optional)

Fig. 1 General workflow of LD Frameworks [12]

c. Silk [22] – The Linked Data Integration Framework

Silk is a Link Discovery framework that is used to generate links between entities from different data sets. It was introduced in the following work "Silk – A Link Discovery Framework for the Web of Data" by J. Volz, C. Bizer, M.Gaedke and G.Kobilarov in 2009[26]. It is based on the Linked Data paradigm exploiting both the data model provided via RDF for structured information and the ability to set RDF links between different data sources. It supports the generation of **owl:sameAs** links as well as other RDF links.

This framework is provided with a special declarative language called Silk-LSL (Link Specification Language) which is used to specify the linking conditions. This language provides the flexibility of the framework. You can specify the conditions that should be satisfied to link two entities from different sources but also the type of RDF link that you wish to apply. Also you have the flexibility to combine various similarity metrics applied in different properties of an entity in order to create link between two entities.

The following picture is indicative of the Silk Workbench. This is the graphical user interface provided by Silk to tune all the parameters mentioned above.

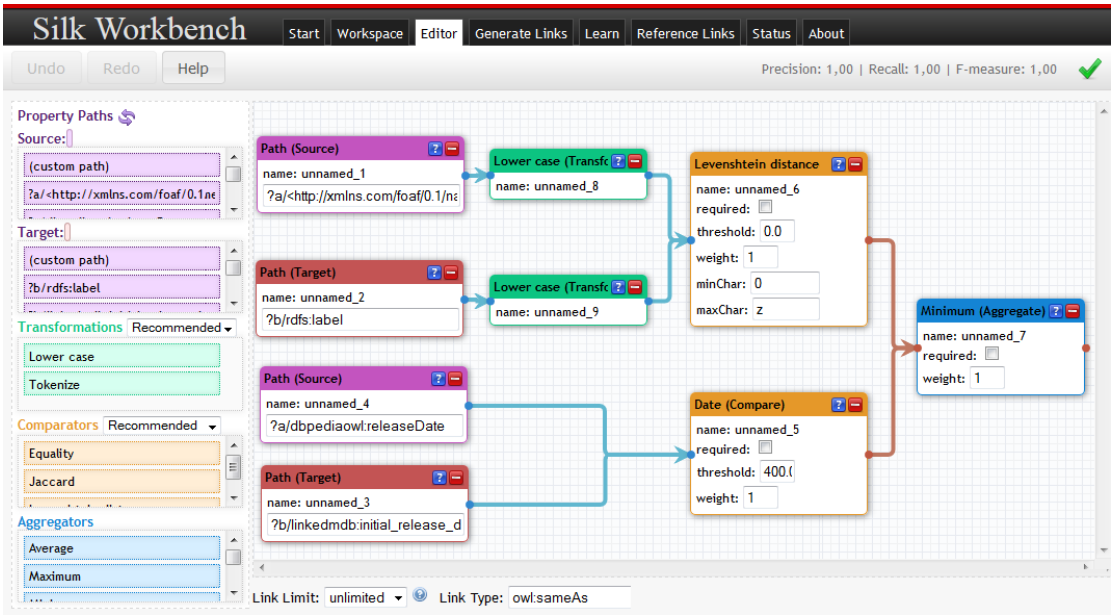


Fig. 2 : Silk Workbench - GUI

d. LIMES [10] - LInk discovery framework for MEtric Spaces

We will briefly refer to the state-of-the-art Link Discovery framework called LIMES that we have mentioned earlier. We will use this framework to compare it with our algorithm in several different experiments and this is the reason that we will cover many more features that this framework has. We have selected this framework instead of Silk because it outperforms Silk in terms of runtime and memory as it was shown in the work that introduced LIMES as a new Link Discovery framework.

LIMES was introduced in the following work "LIMES—A Time-Efficient Approach for Large-Scale Link Discovery on the Web of Data" in 2011. At the beginning LIMES dealt with Link Discovery based on properties that were strings. It could use different similarity metrics for strings like Levenshtein, trigrams, Jaccard, Cosine etc. The first innovation introduced by LIMES was the blocking technique that was based on the creation of exemplars and the second innovation was the filtering technique used, that exploited the triangle inequality property which holds for several string similarity metrics (such as Levenshtein, Jaccard etc) to prune a large number of candidate strings. One year later the framework abandoned the idea of exploiting the triangle inequality property and adopted the algorithms EDJoin[31] and PPJoin[32] to perform string matching tasks, as it was presented in the following work "On Link Discovery using a Hybrid Approach" in 2012[15]. We mentioned these algorithms in a previous chapter as algorithms that are used for solving the "String Similarity Joins" problem. Many new features were added from 2011 such as (1) the ability to perform complex queries using Boolean operators [13] (2) the ability to perform queries using different data types by the integration of the HYPPO and HR3 algorithms [15], (3) the ability to use parallel hardware to perform fast link discovery tasks [16], (4) lastly it implements supervised and unsupervised machine-learning algorithms for finding accurate link specifications. The algorithms include the supervised, active and unsupervised versions of EAGLE[14] and WOMBAT[10]

The following picture shows the graphical user interface used to tune the parameters of LIMES:

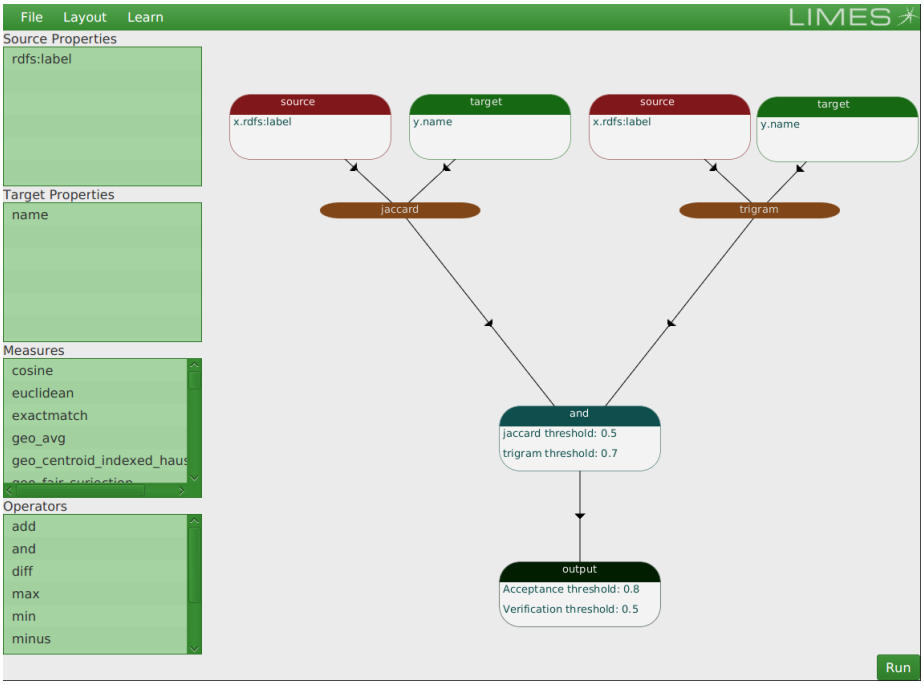


Fig. 3 : LIMES graphical user interface

LIMES framework uses a language which is called LIMES Specification Language (LSL) [10] to create a configuration file that specifies the parameters of the matching task we want to perform. It can work as a standalone tool or as a Java library and using the configuration file you can either access the datasets that will be used as source and target locally or by including the URL of the SPARQL endpoint.

e. String similarity algorithms

There are many string similarity algorithms proposed over the last decades. Based on the method used to quantify the similarity of strings we can divide them into two different categories. The first category is the **Token- based Metrics** and the second category is the **Character- based Metrics** [23].

In the first category the metrics transform the strings into sets of tokens and then they use the set-based similarity metrics to quantify their similarity. There are two types of tokens; those that rely on one or more special characters such as white space, semi-colon etc, so since the string is tokenized depending on the position of the special characters each token has a different length. The second type is called q-grams, where each token has a fixed length. For example, the 5-gram set of “all strings” is {“all s”, “ll st”, “l str”, “ stri”, “strin”, “tring”, “rings”}. If we tokenize using as a special character the white space then we have the following set {“all”, “strings”}. The token-based metrics are suitable for long strings such as essays and documents. For simplicity, each element in the set (token or q-gram) is called a token and we also use string s to denote its corresponding token set. The well-known token-based metrics include Overlap, Jaccard, Cosine, and Dice [31].

In the second category the metrics quantify the similarity of strings based on character transformations. In this category there are many algorithms proposed such Hamming distance, the Levenshtein distance(edit distance), the Damerau-Levenshtein distance and the Jaro-Winkler distance. A comparison of string similarity metrics can be found in “A comparative evaluation of string similarity metrics for ontology alignment” [23]. The main contribution of this work is that it offers a thorough experimental comparison of string similarity algorithms under the same experimental framework and utilizing a variety of authentic datasets.

We will use the Levenshtein distance [8] as a string similarity metric to perform our experiments. This is also called edit distance which is, commonly used as a strings similarity metric. The Levenshtein distance quantifies how similar two strings are to one another by calculating the minimum edit operations needed to align these two strings. The method used to compute the distance is presented in detail in the section where we explain our framework. We are using this metric because (1) we focus on relatively small strings (2) we exploit the triangle inequality property which holds for this metric while for other Character –based metrics (e.g. Jaro-Winkler) this property does not hold.

4. Contribution

This work presents a lossless, time and space efficient approach for large scale link discovery using string similarities. The proposed approach uses the idea of dividing the target data set to blocks by using exemplars. This idea was initially presented by LIMES. Our approach uses a different strategy than LIMES for partitioning the target dataset T to blocks. It creates clusters by utilizing string length and then blocks by utilizing the Levenshtein distance between strings. It combines different filtering and verification methods for string similarities, towards pruning the set of candidate pairs to be considered and reducing the comparisons needed for linking the entities.

The contributions of this work are as follows:

- Optimized blocking method: There is a new proposal in organizing the target dataset T in order to have more pruning power. The two main criteria for the target dataset clustering are two properties of these strings. The first one is the *string length* and the second one is the *Levenshtein distance* in relation to a reference string which is called exemplar string **e**. These two properties are associated with each string as key values of the data structure used. Lastly, we present an indexing method to create a dictionary of string segments per block that correspond to a list of strings that contain this segment.

- Optimized filtering methods: There is a proposal for the application of *length filtering*, *triangle inequality filtering and matching substring filtering*. These three filtering steps guaranty a time efficient output of all the candidate pairs of strings. So, every query uses these three filtering steps to produce a relatively small set of strings for verification. We compare two different implementations of the matching substring filtering in terms of runtime in order to select the most efficient method.

- Optimized verification method: the verification step uses an optimized approach for computing the edit distance between two strings. This approach exploits *prefix pruning, length pruning and early termination technique* to compute efficiently the distance between two strings.

- We have tested our method against LIMES using various datasets of different size containing strings of various lengths. The experimental results show that the optimized version of our algorithm outperforms LIMES in terms runtime and memory footprint.

5. Problem Formulation

Subsequently we refer to the link discovery task as a matching task, given that we aim to compute matching pairs of entities via the computation of joins between strings given as values to key properties of entities in data sources.

a. The Link Discovery problem

MT(S, T, ϑ)

Definition 1 (Matching Task). Given two datasets S (source) and T (target) of entities, a metric m and a threshold $\vartheta \in [0,1]$, the goal of the matching task is to compute the matching pairs (s,t) of all entities $s \in S$ and $t \in T$, such that $m(s,t) \leq \vartheta$.

Q(s, T, ϑ)

Definition 2 (Query). Given one entity s (source) and a dataset T (target) of entities, a metric m and a threshold $\vartheta \in [0,1]$, the goal of a query is to compute the matching pairs (s, t) of all entities $t \in T$, such that $m(s, t) \leq \vartheta$.

Consequently a matching task can be seen as a set of queries.

Since we deal with a matching task considering string similarities, when referring to an entity we refer to an aggregation of string values to key properties of that entity, denoted by s. Thus s denotes the entity and the corresponding string representing that entity.

b. Mathematical framework

Given a set of points in an affine space A*, a function* $m:A\times A\rightarrow R$ *is a metric such that for any* x, y, z *in* A *the following properties are true:*

- $m(x, y) \geq 0$ *(*non-negativity*)*
- $m(x, y) = 0 \Leftrightarrow x=y$ *(*identity of indiscernible*)*
- $m(x, y)=m(y, x)$ *(*symmetry*)*
- $m(x, z) \leq m(x, y) + m(y, z)$ *(*triangle inequality*)*

*The pair (*A, m*) is called a metric space. Although any metric may be used, in this work we use the Levenshtein distance towards comparing strings. It should be noticed however that any distance function (i.e any function that measures the distance or dissimilarity between entities) can be transformed into a normed similarity function.*

The computation of the distance m(s, t) is called a comparison. The computation of the Levenshtein distance between a pair of strings is explained in details in the next paragraphs. Concerning the complexity of the matching task, this is measured by the number of necessary comparisons: A-priori this requires $O(|S|*|T|)$ comparisons. The aim of matching algorithms is to reduce this complexity by applying advanced filtering and verification methods: This reduction is very important for algorithms in order to scale to very large datasets. While the cost of comparisons is usually neglected, this may also be an additional cause of increased complexity, especially when comparing, for instance, large strings: Doing it in a naive way, this will require $|s|*|t|$ operations, for any pair $(s, t) \in S\times T$, where $|.|$ denotes the length of a string.

The triangle inequality (4th property of metric functions mentioned above) has been extensively used for filtering out candidates in many tasks performed in metric spaces. As it is mentioned in [13], the triangle inequality can be used to approximate the distance between two points x, z *in a metric space, when one knows the distances of those points to a third,* reference*, point* y. *This can be done via the inequality*

$$m(x, y) - m(y, z) \leq m(x, z) \leq m(x, y) + m(y, z)$$

These reference points in [13] are called exemplars and are representatives, or samples, of portions of the metric space. According to the above inequality, information on distances from exemplars (e.g. y) can be used to compute lower and upper bounds between points (e.g. x,z) in the metric space. Furthermore, given a threshold ϑ for distances and the lower bound of the m(x,z), it holds that:

$$m(x, y) - m(y, z) > \vartheta \Rightarrow m(x, z) > \vartheta$$

Thus, knowing the distances from exemplars is enough to filter out matching candidates.

Aiming to compute matching pairs of entities via the computation of joins between strings given as values to key properties of entities in data sources, as already mentioned, this work proposes an approach that builds on a basic indexing method, facilitating pruning dissimilar pairs and verifying candidate pairs, effectively. The basic algorithm builds an index for strings in the target set T *, partitioning* T *into portions and identifying exemplars for each portion in an incremental way (i.e. while scanning* T *), and then scans* S *using the index on* T *to compute the results.*

As said, the metric function used to measure the edit distance between two strings s, t *is the Levenshtein distance,. Using a dynamic programming algorithm, a 2d array* M *with* |s|+1 *columns and* |t|+1 *rows is used.* M[i][j] *is the distance between the prefix of* t *with length* i *and the prefix of* s *with length* j*. Initially* M[i][0]=i *and* M[0][j]=j *for* $0 \leq i \leq |t|$ *and* $0 \leq j \leq |s|$ *. Then each matrix cell can be computed by the following equation:*

$$M[i][j]=\min(M[i-1][j]+1, M[i][j-1]+1, M[i-1][j-1]+\gamma)$$

where $\gamma=0$ *if the* i*-th character of* t *is equal to the* j*-th character of* s*, else* $\gamma=1$*.*

While the complexity of the algorithm is O(|s|*|t|), *we can avoid unnecessary computations, given the threshold* ϑ*, by computing only values* M[i][j] *for* $|i-j| \leq \vartheta$*. Doing so, the complexity is improved to* $O(\vartheta * \min(|t|,|s|))$*. Other termination techniques (e.g. those proposed in [9]) may further improve performance.*

c. Notation

We will use the following notation to explain all the methods and algorithms used.

TI : Triangle Inequality

dp: dynamic programming

Lev(s, e): the Levenshtein distance of the two strings s, e

|s|: the number of characters of string s

C: the set of clusters

E: the set of exemplars

E(i): the set of exemplars for length i

|E|: the number of exemplars

|E(i)|: the number of exemplars for length i

S, T: source and target dataset

|S|, |T|: the number of instances of every dataset respectively

floor(x) : returns the smaller integer value of the real number x

ceil(x) : return the larger integer value of the real number x

ϑ: threshold of a matching task

ϑe : threshold used to trigger the creation of a new exemplar

6. Overview of the examined methods

In the next two paragraphs there will be a brief overview of the examined methods. These methods can be divided in three distinct parts: ***the blocking step, the filtering step and the verification step.***

For the ***blocking step*** we introduce a method to divide the target dataset to clusters based on the strings' lengths. Strings t of the target dataset T that have the same length are in the same cluster. We create a set of exemplars per cluster (or per string length) by choosing the most dissimilar strings inside each cluster. Each of these exemplars creates a block of strings. This work examines two main methods to organize each cluster of the target dataset to blocks using the exemplars. The first one is called **First Fit (FF)** approach and the second one is called **Best Fit (BF)** approach. These two methods are thoroughly explained in the paragraph where the creation of exemplars is examined. Also, based on an experimental evaluation we select the **Best Fit** approach, which is more efficient in pruning a relatively larger amount of candidate strings. While the ***blocking step*** takes place before we execute

any query as a preprocessing stage, the ***filtering step*** and the ***verification step*** take place while we execute queries from the source dataset.

The ***filtering step*** consists of three distinct filtering methods that are applied when we execute a query. The ***length filtering*** is based on the strings' lengths. The application of this filter determines a range of clusters that have to be processed further to find the candidate strings matching the query string. The number of strings that is produced as an output is steady and it does not depend on the parameters used. The ***triangle inequality filter*** exploits the **triangle inequality (TI) property.** Inside any cluster of the range of clusters determined, the TI filter determines a relatively small set of candidate strings for matching per exemplar (block). The number of candidate strings determined depends *on the number of exemplars* and on the partitioning scheme that is used to partition clusters to blocks. The last filter used is the **matching substring filter** which further reduces the number of strings that have to be verified.

The third part is the ***verification step***. The verification step calculates edit distances to verify a candidate matching string. In order to make this step efficient we have to apply as many optimizations as possible, so that the complexity of the quadratic **edit distance** dynamic programming algorithm for computing the Levenshtein method is reduced.

The target dataset:

The algorithm begins by determining the cluster of each string of the target dataset T. The main criterion used in the first place is the length of the string. For example, if a target dataset contains strings with 20 different lengths, then our method will create 20 different clusters. Every string of the target dataset needs to be inside the cluster which contains strings, whose length is exactly the same. So, each cluster contains the strings that share the same length. Each cluster has none, one or many exemplars. The first exemplar of a cluster whose length is A, is the first string s_1 from the target dataset that has the given length A. This is how we initialize a cluster. The next exemplar of this cluster is another string s_2 whose length is A, but $Lev(s_1, s_2)$ is greater than the threshold that the user has predetermined for triggering the creation of a new exemplar (ϑe). Inside each cluster, every string of the target dataset is organized based on the Levenshtein distance that is calculated between the string and the exemplars of the cluster. To be more precise we examine the two methods mentioned earlier as First Fit and Best Fit approach. In the first approach the string that we want to organize becomes a member string of an exemplar (block), if the Levenshtein distance is less than the specific threshold ϑe that is predetermined from the user. We call this method "First Fit" because a string will be classified inside the first exemplar satisfying the threshold test. This means that there is a possibility that another exemplar is more similar to the string, that we want to classify. In case there is no exemplar satisfying this threshold criterion, then the new string becomes a new exemplar of the specific cluster. In the second approach *we have to find the edit distance between the string t and all the exemplars of cluster* |t| *and insert the string to the block whose exemplar's distance from t is the minimum*. This is the reason that this method is called "Best Fit".

The source dataset part:

Each string s of the source dataset S corresponds to a query $Q(s, T, \vartheta)$. For each one of these queries we apply three filtering methods. By using ***length filtering*** we specify the

set of clusters (and thus, blocks) that we have to search. By using ***triangle inequality filtering*** we specify for every exemplar, the "area" in a block where candidate strings for matching exist. By using ***matching substring filtering*** we exploit the fact that in order for two strings s, t to match within a threshold ϑ, the string s should have a substring that matches a segment of t, considering that t is divided in ϑ+1 segments. *This property is necessary but not sufficient to answer if the string satisfies the edit distance threshold set by the user*. The filtering techniques used output a small set of strings for each query Q(s, T, ϑ) that have to be further *verified* as matching to the string s, within a distance ϑ.

After the filtering step the last step is *verification*. Verification is based on calculating the Levenshtein distances between each candidate pair <s, t> to see if a string meets the predefined requirements. Since we have to calculate the distance to verify that a string matches any candidate one, we propose and implement several optimizations on how the calculation of edit distance between two strings can be performed efficiently.

7. The Blocking step

a. Exemplar class

The exemplar class has two data members. The first data member is the string of the exemplar. The second data member is the block of the exemplar. The data structure used to represent the ***block*** is a hash map, which has an integer value as a key value and a list of strings as the content of the key value. We have selected the hash map as our data structure to organize the strings inside an exemplar. ***We have made this choice so that the key value of the hash map corresponds to the calculated edit distance between the string and the exemplar***. So, if we want to find all the strings that have distance between Lev(s, e) - ϑ and Lev(s, e) + ϑ, we can retrieve all the lists of strings in constant time.

b. Length Based Clustering

The first algorithm begins by clustering each string of the target dataset. The main criterion used is the length of the string. Every string of the target dataset T needs to be inside the cluster which contains strings, whose lengths are equal. This step will categorize all the strings using the number of characters that a string has. The next idea is to create a set of exemplars per string length. This set of exemplars is a very small subset of all the strings that a cluster contains. They divide the cluster to blocks of strings. So, we can say that each exemplar represents a block of strings. Two questions that rise and they are very important to answer are the following:

1) What is the cost of this pre-computation step?
2) Is it worth paying this cost and in which cases?

The first question is very easy to answer, since we only have to take into account the computation needed to calculate the length of a string. For a string s, the length of the string is a property of class String, which means that the time complexity to retrieve the length is O(1).

The second question does not have an obvious answer. Based on the length of a string we can conclude if this string is worth verifying or not. This pre-computation step organizes the target dataset ***so that in constant time we can retrieve all the strings that have the proper length***. Assuming that we have the query Q(s, T, ϑ) which means that we have a source string s with length |s|, a target dataset T that has M strings organized in N clusters, where N is the number of different lengths of the M strings and ϑ is the given threshold. Then we know that the clusters that we have to take into account are those whose length is |s| – ϑ to |s| + ϑ. This is the first filtering technique that we will use to minimize the number of verifications needed to find all the matches.

c. Creating Exemplars (per length, for every different length)

To create the set of exemplars E for each cluster we use a comparison between a threshold ϑe that the user has predetermined and the Levenshtein distance between the target strings t and the exemplars of a cluster.

Let us consider a cluster whose members are of length A. This cluster could have none, one or many exemplars. *The first exemplar of a cluster whose length is A, is the first string s_1 that we will encounter in the target dataset, that has the given length* $|s_1|$=A. This is how we initialize the set of exemplars of a cluster. The next exemplar of this cluster is another string s_2 whose length is again A, but the Levenshtein distance between s_1 and s_2 is greater than the threshold ϑe that the user has predetermined.

The reason behind the use of this threshold ϑe is that we want to test the behavior of the algorithm when we increase or decrease the number of exemplars per length. We want to see the way the number of exemplars and the partitioning of each cluster, through these exemplars, affects the number of comparisons needed. Theoretically, even one exemplar per length can give us the desired result but we are not sure for the efficiency of this method.

d. First Fit Approach

Inside each cluster, every string of the target dataset is organized based on the Levenshtein distance that is calculated between the string and the exemplars of the cluster. To be more precise a given string ***s*** becomes a member string of an exemplar ***e***, if the Levenshtein distance is less than $|e| - \vartheta e$. We call this method "First Fit" because a string will be classified inside an exemplar *based on the order followed to check if this specific requirement is satisfied*.

This means that there is a possibility that another exemplar is more similar to our string, but in this method we do not care about this fact. In case there is no such exemplar, so that the distance is less than the given threshold, then this string becomes a new exemplar of the specific cluster.

Another fact that is implied is that there is a strong possibility to have strings in the target dataset that are "closer" to exemplars in different clusters (i.e. with different lengths). This means that these exemplars don't have exactly the same lengths but still they are very close so that one may consider them near duplicates.

In case this threshold ϑe is set to zero this means that we will create exactly one exemplar per cluster. This happens because between an exemplar **e** and a string **t** that has the same length ($|t|=|e|$) the distance Lev(e, t) could be less or equal to its length $Lev(e,t) \leq |t|$.

```
Data: Target dataset T, threshold ϑe
Output: N Clusters with a set E(i) of exemplars and their
matching to the instances in T, where i corresponds to a
string length

for t ∈ T do

Set i = |t|
if E(i) = ∅ then
      Set E(i) = E(i) ∪ {t}
else
      for e ∈ E(i) do
            if Lev(t,e)<= ϑe then
                  Add t to the List of Strings in the
                        hashmap of e with key value = Lev(t,e)
            else
                  Set E(i) = E(i) ∪ {t}
            endif

      end

endif

end
```

Alg. 1: Computation of Exemplars and organization of T - First Fit Approach

e. Best Fit Approach

The above method creates exemplars per cluster and puts every string inside each exemplar block using the minimum comparisons (computations of the Levenshtein distance between two strings), but it has some disadvantages. The most important is that inside each cluster, depending on the order that we follow, the first exemplars' blocks are filled with strings but the last exemplars' blocks remain relatively empty. So, this leads to non balanced blocks. To be more precise the comparisons needed to insert every single string inside the data structure that we have created is in the worst case scenario |T|*c, where c is a constant which is a small integer that represents the average number of exemplars that we have created per length. We will show that this scheme leads to poor pruning of candidate strings.

The scheme that we will adopt is called *Best Fit Approach*. According to that, every string of the target dataset is organized based on the Levenshtein distance that is calculated between the string and the exemplars of the cluster it corresponds. To be more precise a given string ***s*** becomes a member string of an exemplar ***e*** under two conditions. The first condition is that the Levenshtein distance of ***s*** to ***e*** is the minimum distance between all the distances of *s* to **E(i)**, where i = |s|. The second condition that has to be satisfied is that the Lev(s, e) has to be less than *ϑe*. In case no exemplar inside the cluster satisfies these conditions, then this string becomes a new exemplar of the specific cluster. Thus, in this "Best Fit" approach a string will be inserted in the block of an exemplar *based on the minimum distance between the string* s *and every* e ∈ E(i)*, where* i =|s|. This scheme leads to a more balanced allocation of the strings in exemplars' blocks.

```
Data: Target dataset T, threshold ϑe
Output: N Clusters with a set E(i) of exemplars and their
matching to the instances in T, where i corresponds to a
string length

for t ∈  T do

Set i = |t|
if E(i) = ∅ then
      Set E(i) = E(i) ∪ {t}
else

      minDistance=|t|

      minIndex = -1
      for e ∈  E(i)  do
            if Lev(t,e) < minDistance then
                  minIndex = index of e

                  minDistance = Lev(t,e)

            endif

      end for

      if(minDistance < ϑe)

            Add t to the List of Strings in the hashmap of e
with key value = minDistance
      else
            Set E(i) = E(i) ∪ {t}
      endif

End for
```

Alg. 2 : Computation of Exemplars and organization of T - Best Fit Approach

8. The Filtering Step

a. Length Filtering

The first filtering method that we will use is based on the length of the strings. For a matching task *MT(S, T, ϑ),* we have arranged the target dataset, so that all the strings that share the same length are in the same cluster. For a given string s from the source dataset S, we execute a matching query Q(s, T, ϑ). We know that all the candidate matching strings are inside the clusters whose length is between |s| – ϑ and |s| + ϑ. Let us assume that we have a string t for which ||s| - |t|| > ϑ. This means that t has either ϑ + 1 less characters or ϑ + 1 more characters than s. In both cases even when we have the best case scenario (s is exactly the same as a substring of t or s has a substring which is exactly the same as t), we need at least ϑ + 1 edit operations (deletes or inserts) to align the strings. Thus, the candidate strings t have a length string for which it holds that ||s| - |t|| <= ϑ. So, we can ignore all the other clusters.

b. Triangle inequality filtering

The second filtering method that we will use is based on triangle inequality in metric spaces.

Let us think every string as a point in a metric space. Each point has a distance from another point based on a string similarity measure, such as the Levenshtein distance, so that the triangle inequality is satisfied.

As a generalization for metric spaces, we can say that the distance between two points x, z in a metric space is less or equal to the sum of the distances between x and a reference point y and the distance between the reference point y and z.

Also there is the reverse triangle inequality which is more useful in our case because it gives us lower bounds.

$$m(x, z) \geq |m(x, y) - m(y, z)| \qquad (1)$$

Any side of a triangle is greater or equal to the absolute difference between the other two sides.

Inequality (1) implies that the distance between two points x, z in a metric space is greater or equal to the absolute difference of the distances between x and a reference point y and the distance between the reference point y and z.

As a consequence of inequality (1) we can conclude that if the difference between two sides that are known is greater than a threshold ϑ, then the other side of the triangle is greater than this threshold T too.

$$|m(x, y) - m(y, z)| \geq \vartheta \Rightarrow m(x, z) \geq \vartheta \qquad (4)$$

We will use the above conclusion for filtering / pruning the candidate strings for verification. This is the exact implementation of the filtering.

```
Data: N Clusters with a set E(i) of exemplars and their
matching to the instances in T, where i corresponds to a
string length, threshold ϑ, source string s
Output: set of candidate strings C

for i = |s| − ϑ  to  |s| + ϑ
      for e ∈ E(i) do
            for key = Lev(s, e) − ϑ to Lev(s, e) + ϑ
                  C = C ∪ {list(key)}
```

Alg. 3 : Computation of candidate matching strings C

The following shape explains what happens when we execute a new query with a source string **s**. In the highlighted area we have all the strings that are candidate strings for verification.

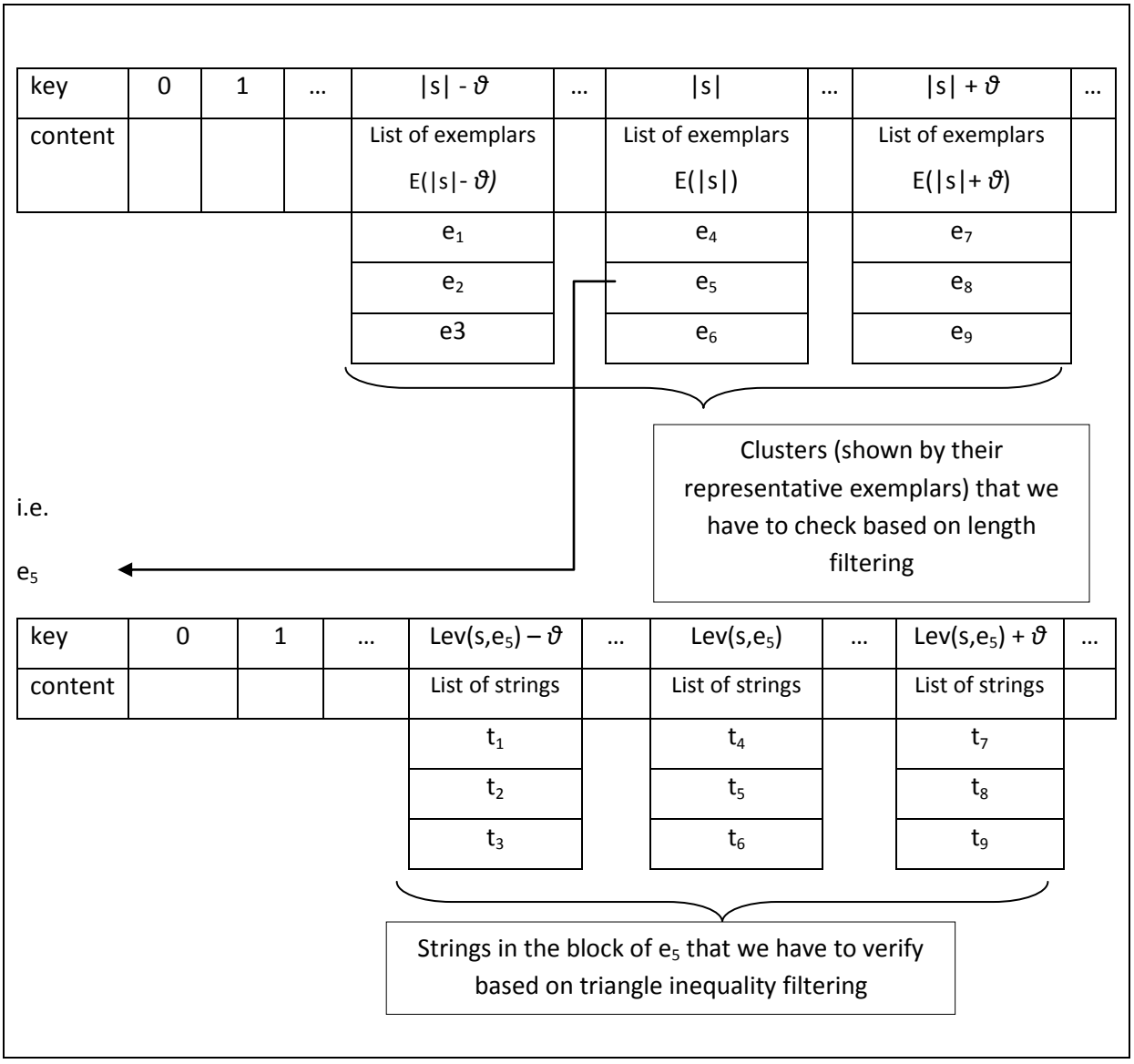


Fig. 4 : A query Q(s, T, ϑ) in the data structure of the target dataset T using Sema

c. Matching Substring Filtering

The third filtering method that we will apply is based on the properties that two strings have to meet in order to have an edit distance which is less or equal that a threshold ϑ. First of all we define a segment of a string as a substring of a given string. Given a string s = “computer” we can partition it to 4 non overlapping segments in many different ways. For example these four segments can be {“co”, “mp”, ”ut”, ”er”} or {“com”, “put”, “e”, “r”} etc.

The following is the lemma used for filtering:

Lemma 1. Given a string t with $\vartheta + 1$ segments and a string s, if s is similar to t within the threshold ϑ, s must contain a substring which matches a segment of t [9].

So, let us assume that we have a string t ∈ T, a string s ∈ S and a given threshold ϑ. Another assumption that we have to make is that |t| is larger than the threshold ϑ. The lemma above states that if we partition t into $\vartheta + 1$ disjoint segments (so that each segment has at least one character) then, in order to satisfy the condition Lev(s, t) < ϑ, there has to be at least one substring of s which matches a segment of t. Subsequently we can adopt this filtering method as follows: A string is a candidate string for verification if and only if there is a substring of s that matches a segment of t, from the t’ s $\vartheta + 1$ disjoint segments.

Partition scheme

There are many partition schemes that we can adopt in order to create the $\vartheta + 1$ disjoint segments of t. An effective scheme is able to reduce the number of candidate pairs and improve the performance of our algorithm. If our partition scheme leads to short segments then we lose pruning power. This is due to the fact that it is more likely to find matching substrings for a segment which is short. At the same time if we try to create large segments of a string then we are creating short segments, too. Consider the example of string “computer” that we have already used. Obviously, the partitioning to {“co”, “mp”, ”ut”, ”er”} has 4 segments with the same length, while the partitioning to {“com”, “put”, “e”, “r”} also has 4 segments but two of them have three characters and two of them have just one character. The second set of segments will not prune strings that just have an “e” or an “r” as a character. Another idea that we could adopt is to find rare characters or substrings and rely on this information to partition the string t. On the other hand, in order to create something like this we have to sacrifice again runtime. So, based on these observations the partition scheme that we adopt is an ***even-partition scheme***, which is very simple to apply and it leads to a balanced partitioning so that there are almost equal segments. To be more precise every segment will have either floor(|t|/ (ϑ + 1)) or ceil(|t|/ (ϑ + 1)) length.

This is the exact implementation of the filtering.

```
Function MatchingSubstringToSegment(t,s)
Data: target string t, source string s
Output: Boolean
d = |t| / (ϑ +1)
m = |t| % (ϑ +1)
 for ( i =0; i < (ϑ +1)-m; i++)
                if(s.contains ( t.substring(i*d, (i+1)*d) )
                  return true
 for(i = (ϑ +1)-m ; i < (ϑ +1) ;  i++)
                if(s.contains( t.substring(i*d, (i+1)*d+1))
                  return true
return false
```

Alg. 4 : Determining if t is a candidate string

```
Data: set of candidate strings C, source string s
Output: set of candidate strings C′
for t ∈ C
        if (MatchingSubstringToSegment(t,s))
            C′ = C′ ∪ {t}
        Endif
end
```

Alg. 5 : Computation of all candidates matching strings C′

The above algorithms are implemented in the first version of our program called Sema which we will test against LIMES. An optimized version of our method is called OptSema.

d. OptSema - Optimized application of Matching Substring Filter

The algorithm shown above is a rather naïve application of the lemma presented in the previous paragraph. For a query Q(s, T, ϑ) length filtering and triangle inequality filtering leads to a subset T′ of candidate strings. For every single target string t of the set T′ we have to partition the target string t to ϑ+1 segments and find if there is at least one matching substring of the source string s. The complexity of this operation is O(|s|*|t|/(ϑ+1)). Instead of using this expensive operation for every single query that we need to perform we could prepare the target dataset T to be able to find the strings that have a matching segment more efficiently by using the proper data structure. The idea that we will present is based on the creation of an index of segments for the target dataset T. This index will be implemented using a hashmap that has string segments as the key values and as content they consist of the list of strings that contain that segment. Using this data structure we are able to retrieve all the strings that contain a segment in constant time.

Changes made in the exemplar class

Every exemplar string forms a block of strings and in this block every string is classified based on its edit distance with the exemplar. Now we will add a new data structure in each exemplar that will hold an index of segments for the strings that the exemplar contains. To create this index of segments we will utilize the even partition scheme that we mentioned earlier. Every string classified inside an exemplar will be partitioned to segments. Every segment is a new smaller string than the original which could be a part of other strings too, contained in the block of the exemplar. Using this segment as an index we will build the list of strings that have this segment. To create this index we will use a hash map that has a string as a key value and as content the strings that have this string as a segment.

This is the exact implementation of the creation of segments for a target string t.

```
Function targetSegments(String t)
Data: target string t, threshold ϑ
Output: List of Strings
List is a list of Strings
int d = |t| / (ϑ +1);
int count = |t| % (ϑ +1);
     if(d!=0){
         int i = 0;
         for (i =0; i < (ϑ +1)-count; i++) {
               List.add(t.substring(i*d, (i+1)*d));
          }
          for(i = (ϑ +1)-count ; i < (ϑ +1) ; i++) {
               List.add(t.substring(i*d, (i+1)*d+1));
         }
     }
     else{
          List.add(t);
     }
return List;
}
```

Alg. 6 : Creating the list of segments of target string t

The following shape explains what happens when we execute a new query with a source string **s**. In the highlighted area we have all the strings that are candidate strings for verification.

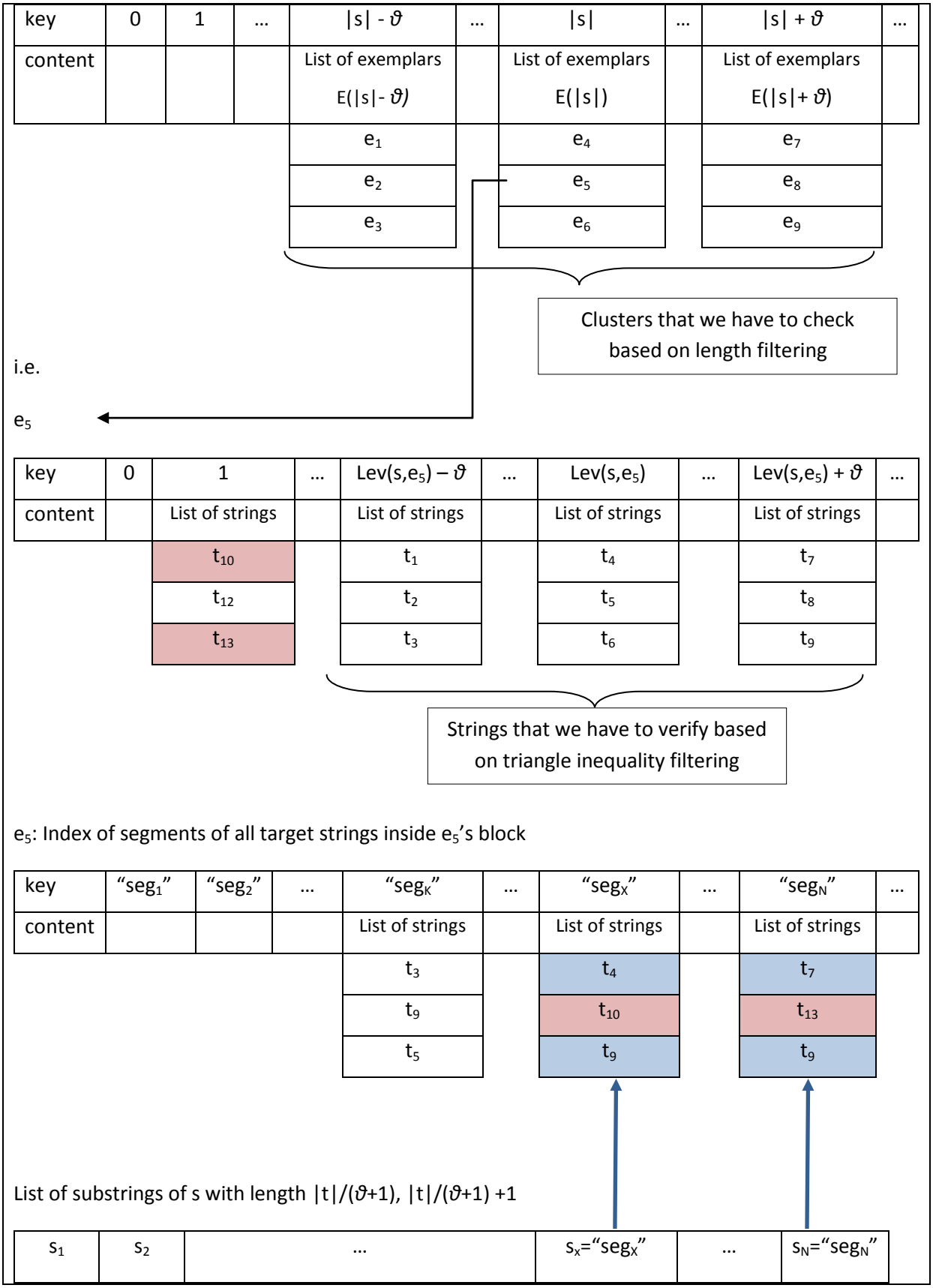


Fig. 5 : A query Q(s, T, ϑ) in the data structure of the target dataset T using OptSema

As shown in Fig. 5 when we execute a query Q(s, T, ϑ) we create a list of all the possible substrings of a source string that have length $|t|/(\vartheta+1)$ and $|t|/(\vartheta+1)+1$. **In approximately linear time O(|s|) we are able to retrieve the subset of strings T' of the target data set T that have a segment that matches a substring of s**. **In approximately constant time we are able to check if a string t of the subset T' belongs to the subset of strings that the Length and TI filtering produces.** ***In other words our main task now is to find the intersection of two subsets of T, the subset produced from applying the Length and TI filtering and the subset produced from the Matching Substring filtering***. From Fig. 5 we can see that string t_{10} has a segment that matches a substring of s, but we can prune it since it does not satisfy the triangle inequality property. On the other hand, if we have a string like t_1 which satisfies the triangle inequality but there is no segment of t_1 that matches a substring of s then again we can prune the string.

The above algorithms are implemented in the second version of our program called OptSema. We will test it against LIMES and the naïve version for implementing matching substring filter called Sema.

9. The Verification Step

a. Naive approach for calculating distances

The edit distance algorithm answers to the following problem:

Given a pair of strings s and t and a set of edit operations (insertion, deletion, substitution) that are allowed with a given cost for each one of them, find the edit operations needed to align s to t, with the minimum cost.

So, in our case the minimum cost calculated by the edit distance has to be less or equal to ϑ.

So, our first approach to verify a pair of strings is to calculate the Levenshtein distance or edit distance and see if the condition is satisfied. The computation of the edit distance uses the dynamic programming (dp) paradigm which divides a problem into sub problems. For each of these sub problems dp uses a memo to store their solution and based on these solutions the algorithm finds the solution to the next sub –problems. The method begins by determining the base cases which are these instances of the problem that have an obvious solution.

In our case the mathematical background for implementing dp to find the edit distance is the following:

There are three **edit operations** that are allowed to align these two strings.

1. Insert a character
2. Delete a character
3. Substitute a character

Each one of these operations has a cost that we have to pay to perform them. This cost is added and the total cost is the **edit distance**. There is a variation of the Levenshtein distance where each operation has a different cost. In our case we consider that every edit operation has the same cost, which can be equal to 1.

Given a pair of strings s and t, and i, j pointers, where s_i corresponds to the substring from character 0 to i_{th} character of s and t_j corresponds to the substring from character 0 to j_{th} character of t, then:

Base cases:

$Opt_{i0} = \sum_{k=1}^{i} 1$ where $1 \leq i \leq |s|$

$Opt_{0j} = \sum_{k=1}^{j} 1$ where $1 \leq j \leq |t|$

Recursive formula:

$$Opt_{ij} = \begin{cases} Opt_{i-1,\,j-1} & \text{if } s_i = t_j \\ \min \begin{cases} Opt_{i-1,\,j} + 1 \\ Opt_{i,\,j-1} + 1 \\ Opt_{i-1,\,j-1} + 1 \end{cases} & \text{if } s_i \neq t_j \end{cases}$$

We will perform a calculation of an edit distance using s="COMPUTERS" and t="CIOMUTES". We have selected these two strings because the edit operations needed to align them involve all three edit operations that are available.

The following table is an example of the computation using the above formulas.

	i	0	1	2	3	4	5	6	7	8	9
j	Opt	null	C	O	M	P	U	T	E	R	S
0	null	0	1	2	3	4	5	6	7	8	9
1	C	1	0	1	2	3	4	5	6	7	8
2	I	2	1	1	2	3	4	5	6	7	8
3	O	3	2	1	2	3	4	5	6	7	8
4	M	4	3	2	1	2	3	4	5	6	7
5	U	5	4	3	2	2	2	3	4	5	6
6	T	6	5	4	3	3	3	2	3	4	5
7	E	7	6	5	4	4	4	3	2	3	4
8	S	8	7	6	5	5	5	4	3	3	3

Fig. 6 : Step by step execution of the dp algorithm for edit distance computation

The base cases are highlighted with green color. The distance is the value of cell Opt[8,9] = 3. This is the optimal solution to align the strings in the sub problem / instance we are solving.

The pseudo code for the Levenshtein distance is the following:

```
int Levenshtein_Distance (string s[1..m], string  t[1..n])
   // d is a table with m+1 rows and n+1 columns
   int d[m+1, n+1]
   for i from 0 to m+1
       d[i, 0] = i
   for j from 0 to n+1
       d[0, j] = j
   for i from 1 to m
       for j from 1 to n
       {
           if s[i] = t[j] then
                cost = 0
           else
                cost = 1
           d[i, j] = minimum(
                                d[i-1, j] + 1,     // deletion
                                d[i, j-1] + 1,     // insertion
                                d[i-1, j-1] + 1   // substitution
                            )
       }
    return d[m+1, n+1]
```

Alg. 7 : Computation of the Levenshtein distance

The time complexity of the dynamic programming algorithm is O(|s|*|t|).

b. Optimized verification Length – aware verification

Simple early termination

To organize the target dataset we cannot avoid computing the edit distance of the strings of the target dataset with the corresponding exemplars. The reason is that we have to position each string in the hash map of an exemplar based on the edit distance between the string and the exemplar. This distance is used as the key value of the hash map. It is worth paying the cost, since it is a nice way to take advantage of the triangle inequality property. By organizing strings like this, we can retrieve all the strings that satisfy the Triangle Inequality property in O(1). On the other hand when we are executing queries from the source dataset, then we are not interested on the *exact distance* between the source string and the target strings. We are interested to find out if a source string and a target string are "closer" than a threshold ϑ. This implies that when we verify a candidate target string we could quickly exclude it as a matching pair if we can deduce that the string has a distance greater than the threshold ϑ.

To simplify the analysis we will use the same example as before when talking about the Levenshtein distance. Let us assume that string t is "COMPUTERS", string s is "CIOMUTES" and the given threshold $\vartheta = 1$.

The algorithm begins by calculating the base cases and continues by calculating each row from the left to the right side of the 2d array. For the Opt_{44} the value is already 2 and since there are characters added on the right while we increment i, then the optimal distance increments too. For the next row Opt_{53} is already 2 and it decrements from 5 to 2, while we add characters by incrementing i. The next computation selects the minimum between the three options that are highlighted with orange color. So since there is a mismatch the minimum is $Opt_{43} + 1 = 2$. *From now on we are sure that every value that we calculate can be greater or equal to the values that are already calculated for the previous sub problems. This is a direct consequence of the recursive formula used to calculate every next optimal solution. So the algorithm can terminate earlier, without having to calculate all the values that are highlighted with dashed cells.*

	i	null	C	O	M	P	U	T	E	R	S
j		0	1	2	3	4	5	6	7	8	9
null	0	0	1	2	3	4	5	6	7	8	9
C	1	1	0	1	2	3	4	5	6	7	8
I	2	2	1	1	2	3	4	5	6	7	8
O	3	3	2	1	2	3	4	5	6	7	8
M	4	4	3	2	1	2	3	4	5	6	7
U	5	5	4	3	2	2	2	3	4	5	6
T	6	6	5	4	3	3	3	2	3	4	5
E	7	7	6	5	4	4	4	3	2	3	4
S	8	8	7	6	5	5	5	4	3	3	3

Fig. 7 : Early termination technique

c. Length pruning

The next improvement that we can implement is based on length pruning. Since the threshold is 1, every instance of the problem involves i characters from the first string and j characters from the second string. So, in this case we have to calculate only the instance d[i,j], where |i-j| is less or equal to ϑ. This is because for every other instance for which |i - j| > ϑ *we conclude that* d[i, j] > ϑ.

	i	null	C	O	M	P	U	T	E	R	S
j		0	1	2	3	4	5	6	7	8	9
null	0	0	1	2	3	4	5	6	7	8	8
C	1	1	0	1	2	3	4	5	6	7	8
I	2	2	1	1	2	3	4	5	6	7	8
O	3	3	2	1	2	3	4	5	6	7	8
M	4	4	3	2	1	2	3	4	5	6	7
U	5	5	4	3	2	2	2	3	4	5	6
T	6	6	5	4	3	3	3	2	3	4	5
E	7	7	6	5	4	4	4	3	2	3	4
S	8	8	7	6	5	5	5	4	3	3	3

Fig. 8 : Length pruning technique

So there is no need to compute these values. So in other words we only have to compute only the values that are inside the orange cells. This optimization improves the complexity of the algorithm from O(|s|*|t|) to O((2*ϑ +1)*min(|s|,|t|).

d. Improving length pruning

We have another useful observation that leads to the next optimization. From the 2d - table we can deduce that d[2,2] instance aligns “CI” to “CO “ with 1 edit operation, and the remaining substrings are “OMUTES” and “MPUTERS” that need at least one edit operation to align based on their length difference. This means that we don’t need to compute this value too since we know that it has exceeded the threshold ϑ. This method is called *length-aware verification* method [9]. Let Δ= |s|-|t| (assuming that |s| is greater that |t|) and Δ is less or equal to ϑ *(if not then Lev(s, t) would be greater than* ϑ*)*.

For each row of the 2d array d, we have to compute d[i,j] for

i -floor((ϑ-Δ)/2) *<= j <= i+floor((ϑ+Δ)/2).*

This improves the complexity to O((ϑ + 1)*min(|s|, |t|). So in our case Δ= 9-8=1 => i <= j <= i + 1. The optimization that we have achieved is shown by the deleted cells:

	i	null	C	O	M	P	U	T	E	R	S
j		0	1	2	3	4	5	6	7	8	9
null	0	0	1	2	3	4	5	6	7	8	9
C	1	1	0	1	2	3	4	5	6	7	8
I	2	2	1	1	2	3	4	5	6	7	8
O	3	3	2	1	2	3	4	5	6	7	8
M	4	4	3	2	1	2	3	4	5	6	7
U	5	5	4	3	2	2	2	3	4	5	6
T	6	6	5	4	3	3	3	2	3	4	5
E	7	7	6	5	4	4	4	3	2	3	4
S	8	8	7	6	5	5	5	4	3	3	3

Fig. 9 : Optimized Length pruning technique

e. Optimized verification based on early termination/prefix pruning

By observing again the way that the optimal distances are calculated in the 2-dimensional array we can propose another optimization of the algorithm. This optimization is called ***prefix pruning***. Based on the discussion from the first optimization proposed we can observe that after d[3,*] there is no chance that the strings have a distance which is less than ϑ. This is because we have two calculated values which are shown in blue color d[3,3], d[3,4] which are greater than ϑ = *1, thus we can stop the algorithm earlier because we are sure that the distance is greater than* ϑ.

	i	null	C	O	M	P	U	T	E	R	S
j		0	1	2	3	4	5	6	7	8	9
null	0	0	1	2	3	4	5	6	7	8	9
C	1	1	0	1	2	3	4	5	6	7	8
I	2	2	1	1	2	3	4	5	6	7	8
O	3	3	2	1	2	3	4	5	6	7	8
M	4	4	3	2	1	2	3	4	5	6	7
U	5	5	4	3	2	2	2	3	4	5	6
T	6	6	5	4	3	3	3	2	3	4	5
E	7	7	6	5	4	4	4	3	2	3	4
S	8	8	7	6	5	5	5	4	3	3	3

Fig. 10 : Combination of prefix and length pruning

So the main conclusion is that we can stop in row d[3,*]. Can we do better than this? What can we say about the previous row which is d[2,*]? In this row one computation d[2,3] is greater than ϑ but d[2,2] is equal to ϑ. So if we think about it we need d[2,2] = 1 to align "CO" and "CI" and then we have to align "OMUTES" which has 6 characters to "MPUTERS" which has 7 characters and we need at least one more edit operation. This means that we can stop our algorithm even before d[3,*], in row d[2,*] by computing the sum of the calculated value and the expected value based on the length difference of the substrings remaining. So, we can stop the computation if for a row i it holds that d[i,*] + ||s|-i |-||t|-*| > ϑ. For both Sema and OptSema we have used the combination of all the techniques shown above to get the best runtime, when verifying a candidate string.

10. Experimental evaluation

The next section shows the experimental evaluation of the above methods in the following order. We evaluate “First Fit” and “Best Fit” based on the comparisons that we need to do in order to reach a result. We show the results and we adopt the better of these two methods. The machine used for the experiments is a

Pentium® Dual-Core CPU T4500 @ 2.30Ghz 2.30Ghz RAM 4.00GB, 64-bit OS

a. Data sets used

The data sets that we have used for our experiments are triples from Ariadne http://www.ariadne-eu.org/ that we have processed. To simplify the process we have distinguished only the triples with the property “title”. We further processed the dataset so that our dataset does not have duplicate URIs and the length of the string value would vary from 10 to 20, from 20 to 30 characters and from 30 to 40 characters. Then we have created different files using different number of triples per file. We will use these datasets to perform experiments to measure the pruning power of our methods, to support further what is theoretically stated and documented in the previous paragraphs for the efficiency of our algorithms and to compare in terms of runtime and memory our algorithms with a state-of-the-art LD framework that we have presented earlier called LIMES.

b. First Fit vs Best Fit

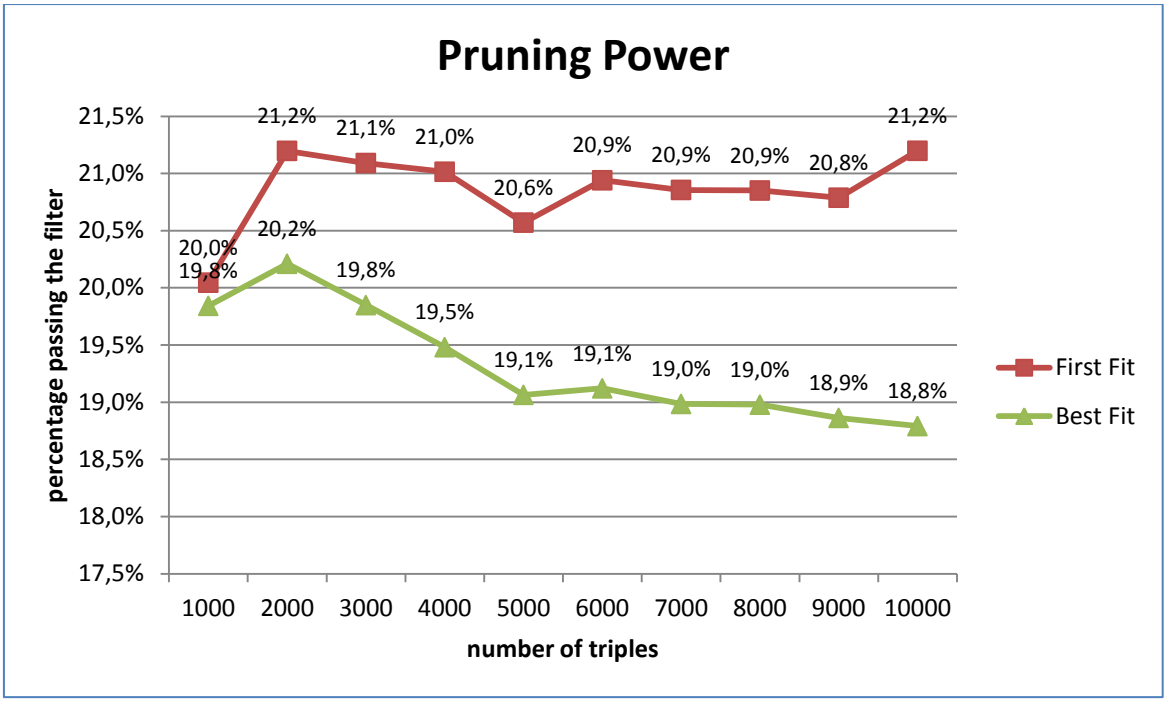
We have tested the two methods for creating blocks inside a cluster, that are mentioned in a previous paragraph of this work, in order to select the most efficient one. We have used data sets containing strings from 1 to 10 characters. The size of the data sets begins from 1000 triples incrementing by 1000 to 10000 triples. The results are shown in the following table:

size	naïve	Exemplars	First Fit	FF pruning power	Best Fit	BF pruning power	difference
1000	1000000	77	200455	20,0%	198430	19,8%	2025
2000	4000000	92	847923	21,2%	808435	20,2%	39488
3000	9000000	106	1898234	21,1%	1786402	19,8%	111832
4000	16000000	111	3362485	21,0%	3116953	19,5%	245532
5000	25000000	122	5142627	20,6%	4766106	19,1%	376521
6000	36000000	125	7538683	20,9%	6883601	19,1%	655082
7000	49000000	130	10219565	20,9%	9302606	19,0%	916959
8000	64000000	140	13344899	20,9%	12146182	19,0%	1198717
9000	81000000	143	16838606	20,8%	15278929	18,9%	1559677
10000	100000000	146	21198689	21,2%	18791190	18,8%	2407499

Table 1: Comparison between First Fit method and Best Fit method

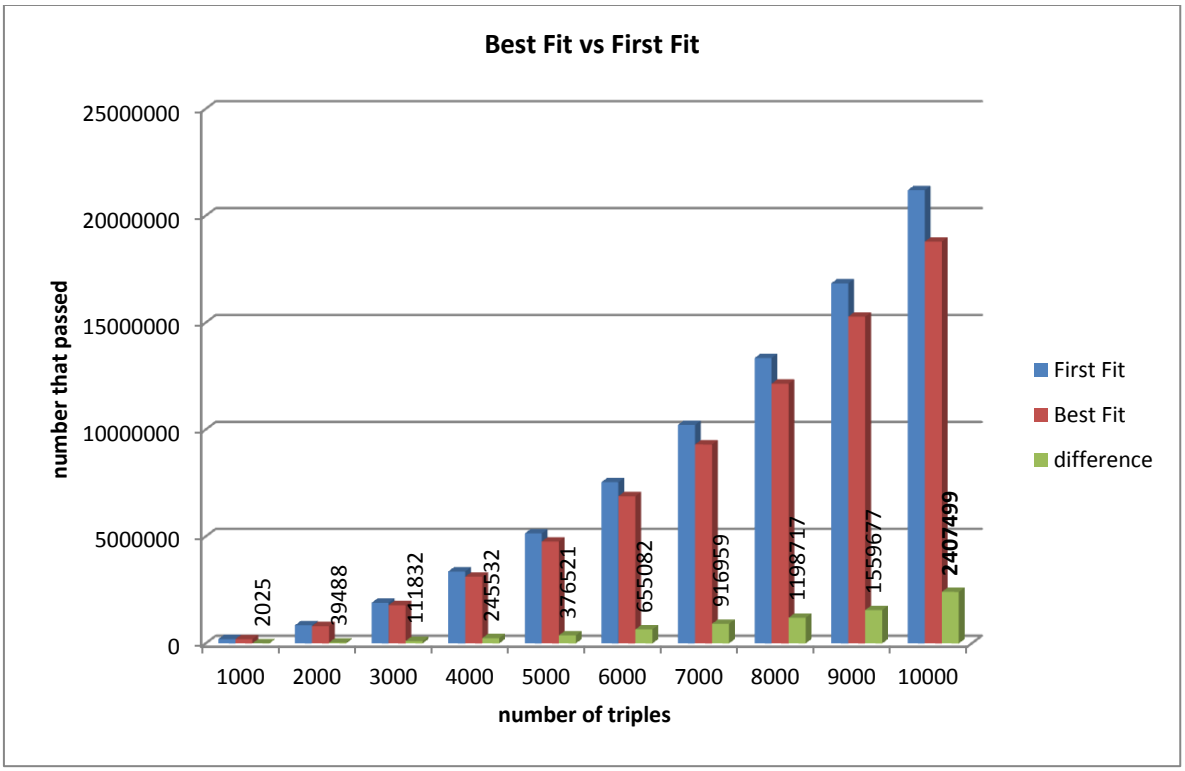
The column “size” contains the number of triples used per experiment. The column “naïve” contains the number of comparisons that would be needed in the case that a naïve algorithm $O(N^2)$ was used to find the matching pairs. The column “Exemplars” contain the exact number of exemplars that were created per experiment. “First Fit” column contains the total number of comparisons (distance computations) needed after the application of the Length and Triangle Inequality filtering. “FF pruning power” column holds the quotient of number of FF comparisons over the naïve number of comparisons represented as a percentage. “Best Fit” column contains the total number of comparisons needed after the application of the Length and Triangle Inequality filtering. “BF pruning power” column holds the quotient of number of BF comparisons over the naïve number of comparisons represented as a percentage. The column “difference” contains the difference in the numbers of comparisons needed between the two methods in absolute numbers (FF number of comparisons – BF number of comparisons).

The graph below offers a graphical representation of the pruning power in percentages that the two methods have depending on the number of triples of datasets used.



Graph 1 : Pruning power between FF and BF

The following graph shows the difference between the two methods in absolute numbers.



Graph 2 : Difference of comparisons needed between FF and BF

The x- axis has the number of triples of the data set used, while the y-axis shows the number of comparisons needed in absolute numbers. We can observe that the difference in comparisons needed, between the two methods, increases as the size of the dataset

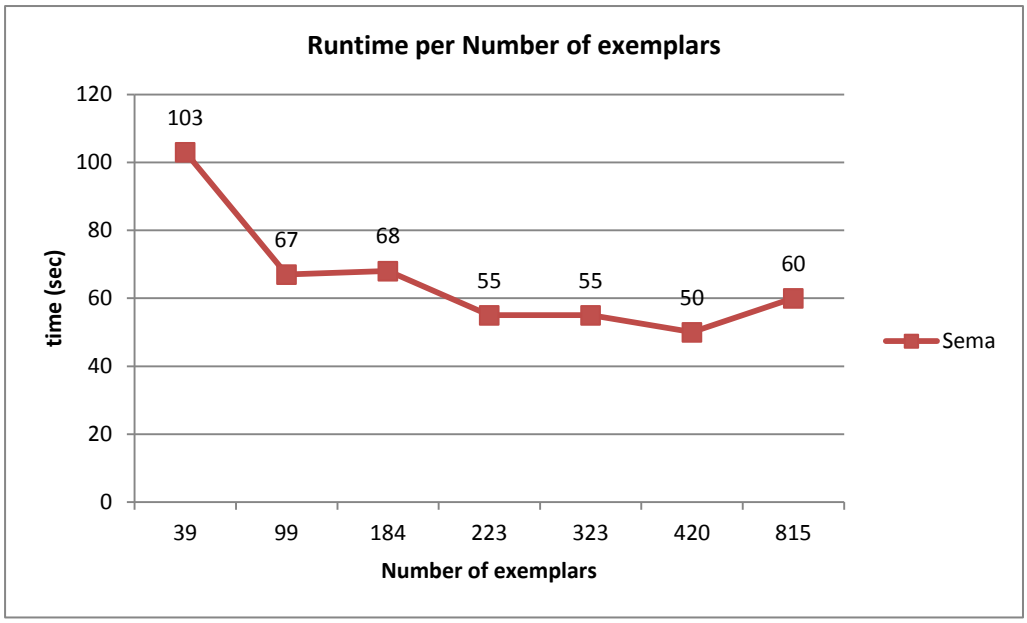
increases. For example in the case of 10000 triples the FF method needs approximately two and a half million more comparisons than the BF method to finish the matching task.

We see that, in terms of comparisons needed to complete a matching task, the BF method outperforms the FF method. So this is the method that we have selected as our main method for creating blocks in each cluster.

c. The impact of the number of exemplars used

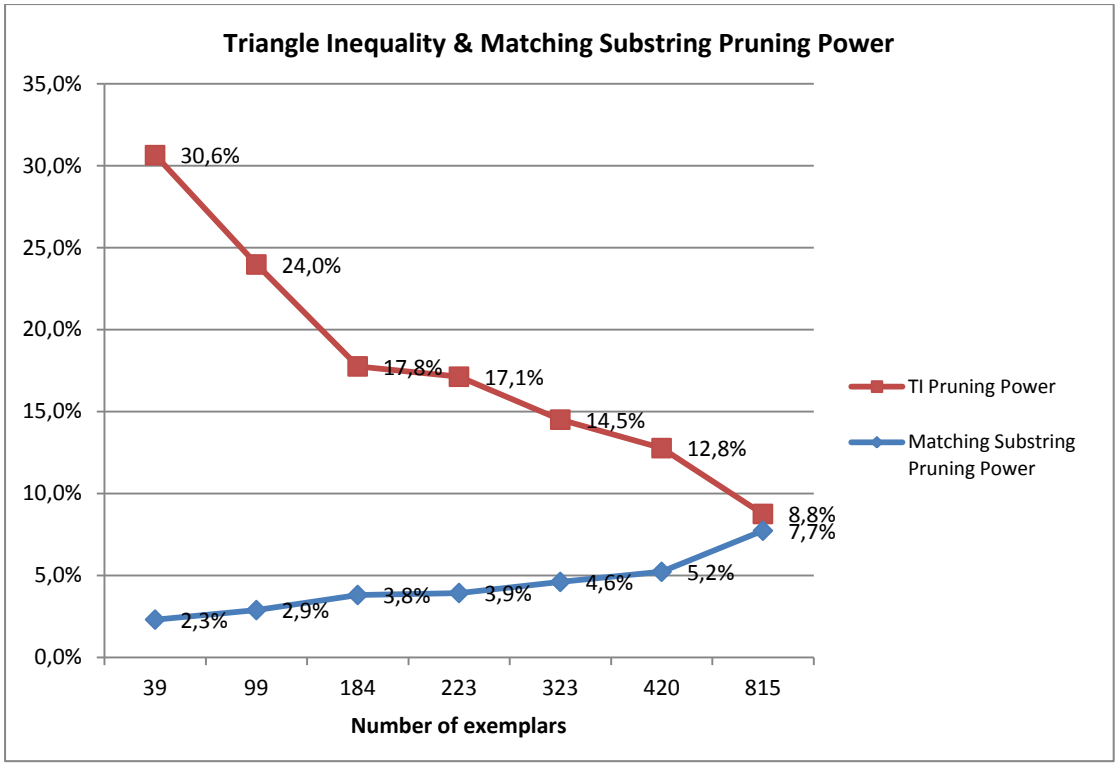
Testing Sema

We have tested Sema using 20000 strings with lengths that vary from 10 to 20 characters. The following experiment uses different number of exemplars to divide the target dataset T and on each case we measure the pruning power and the runtime of the "Best Fit" approach. The goal of this experiment is to measure the impact of the different number of exemplars in terms of comparisons needed and runtime and to see if there is a number of exemplars that we have to use depending on the size of the target dataset T, to optimize the performance of our algorithm. The following chart shows the runtime of our implementation depending on the number of exemplars.



Graph 3 : Sema runtime for different number of exemplars

From the above chart we conclude that we get better results as we increase the number of exemplars. But there is a number of exemplar above which we get the opposite result and the runtime increases. We can conclude that there is an optimal number of exemplars that we have to use to achieve a better runtime. So, to get better results we have to tune the parameters used to run our algorithm.



Graph 4 : Pruning power of TI and MS filtering

From the chart above we can see that while the pruning power of the triangle inequality filter increases pruning power the other filter loses pruning power. If we tune the number of exemplars to 39 then 31% of strings pass the TI filter. After this we apply the Matching Substring filter and we get only 2.3% of the strings that need verification. If we tune the number of exemplars to 815 only 8.8% of strings pass the TI filter. With the application of the Matching Substring we get 7.7% of these strings that need verification.

The following table shows the experimental results.

Naïve	No of exemplars	No of comparisons after TI filtering	TI Pruning Power	No of comparisons for verification after MS filtering	MS Pruning Power
4,00E+08	39	122590951	30,6%	2824362	2,3%
4,00E+08	99	95904909	24,0%	2762912	2,9%
4,00E+08	184	71027630	17,8%	2693435	3,8%
4,00E+08	223	68504819	17,1%	2679303	3,9%
4,00E+08	323	58027326	14,5%	2650476	4,6%
4,00E+08	420	51076663	12,8%	2633775	5,2%
4,00E+08	815	35008353	8,8%	2617074	7,7%

Table 2 : Pruning power of TI and MS for different number of exemplars

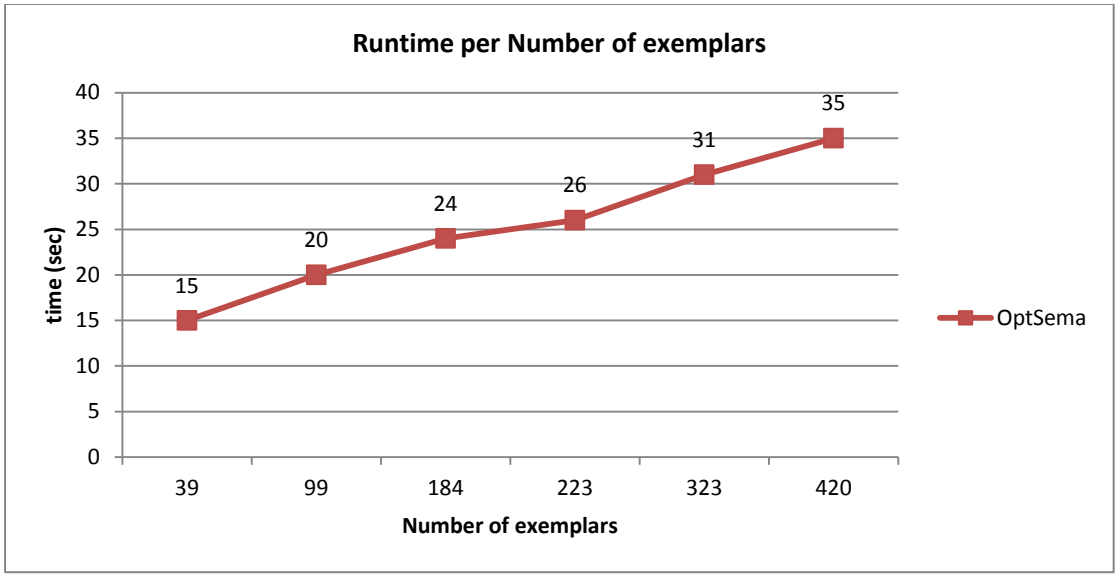
The column “Naïve” contains the total number of comparisons needed if we used the naïve version for matching the two data sets. The column “No of exemplars” contains the number of exemplars that we used per experiment. The column “No of comparisons after TI filtering” contains the number of comparisons that we need after the application of the Triangle Inequality filtering. The column “TI Pruning Power” contains the quotient of the number of comparisons after TI filtering over the number of comparisons of the naïve version. The column “No of comparisons for verification after MS filtering” contains the

number of comparisons that we need after the application of the Matching Substring filtering. The column "MS Pruning Power" contains the quotient of the number of comparisons after MS filtering over the number of comparisons of the naïve version.

We observe that in the case where the exemplars are 39 the number of comparisons (marked with yellow) is enormous after the application of TI filter. The application of the Matching Substring filter then results to rather small number of strings that need verification. Now let us compare the above case with the case of 420 exemplars (marked with green). We can see that we have less than half comparisons than we needed before based on the TI filter. But after Matching Substring filter the difference between the numbers of strings that need verification is quite small (2824362-2633775=190587).

Testing OptSema

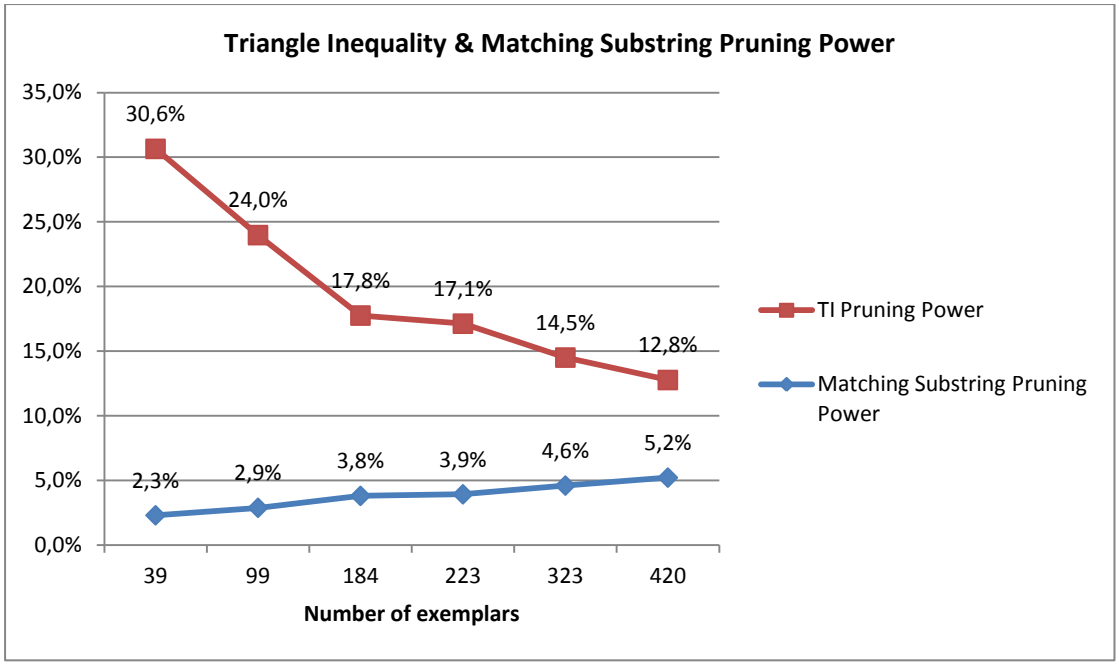
We have repeated the same experiment using the same input file and number of exemplars using the optimized version of our algorithm OptSema. The following chart shows the runtime of our implementation depending on the number of exemplars.



Graph 5 : OptSema runtime for different number of exemplars

Obviously we get better results with as fewer exemplars as possible. A question that we have to answer is why this is happening.

The chart below shows that we have exactly the same behavior in terms of the number of comparisons needed as before.



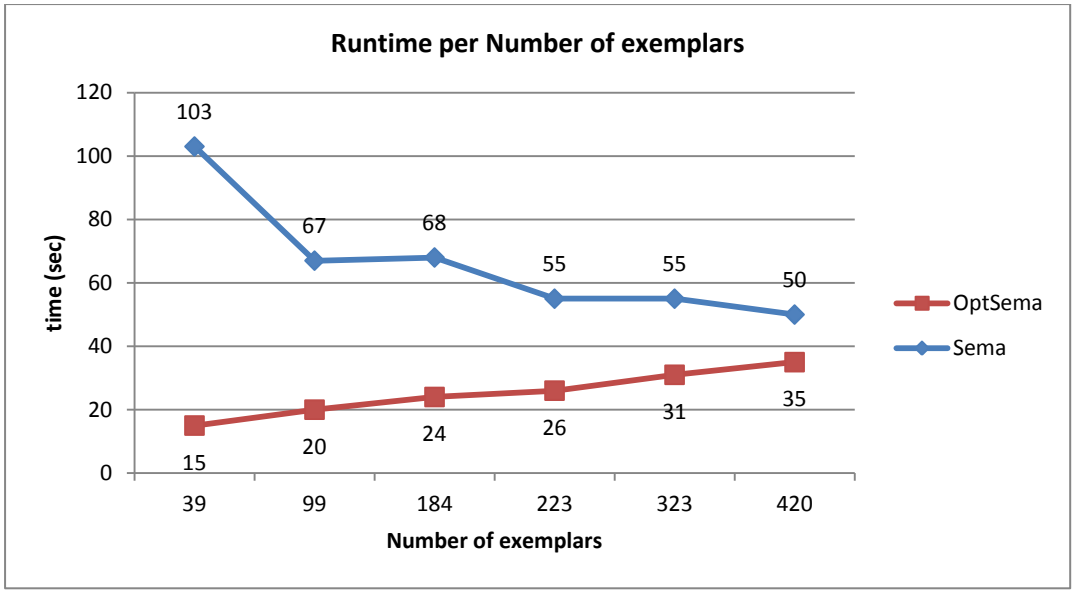
Graph 6 : Pruning power of TI and MS filtering

The table below explains in more detail our findings.

Naïve	No of exemplars	No of comparisons after TI filtering	TI Pruning Power	No of comparisons for verification after MS filtering	MS Pruning Power
4,00E+08	39	122590951	30,6%	2824362	2,3%
4,00E+08	99	95904909	24,0%	2762912	2,9%
4,00E+08	184	71027630	17,8%	2693435	3,8%
4,00E+08	223	68504819	17,1%	2679303	3,9%
4,00E+08	323	58027326	14,5%	2650476	4,6%
4,00E+08	420	51076663	12,8%	2633775	5,2%

Table 3 : Pruning power of TI and MS for different number of exemplars

If we check Table 2 and Table 3 we got **exactly the same results**. Now we have to answer why we get better runtime using OptSema than with Sema.



Graph 7 : Comparison of runtime between OptSema and Sema

The difference is justified based on the algorithm used for the application of the Matching Substring filter. The optimized version OptSema leads ***to very efficient and quick pruning*** of the set of strings that is produced from the TI filter. So even though we have to check huge numbers of strings our method can prune quickly these strings and produce a relatively small set of strings for verification.

d. Comparison for strings with 10 – 20 characters

We have tested LIMES, Sema, OptSema using strings with lengths that vary from 10 to 20 characters. The datasets used begin from 2000 triples, they increment by 2000 until they reach 20000 triples. Based on our previous observations in the case of Sema we have executed the algorithm for different number of exemplars and we have stored the best results. For OptSema we have used the minimum number of exemplars needed to execute the experiment. There is a need for a brief explanation of the threshold used. LIMES uses the following formula to calculate the number of edit operations $\vartheta = 1/(1+d)$ and d is the number of edit operations. So this means that when the threshold is 1 this means that the number of edit operations is 0. If the threshold is equal to a value that belongs to (0.5, 1] then again d = 0. So we have the following conditions:

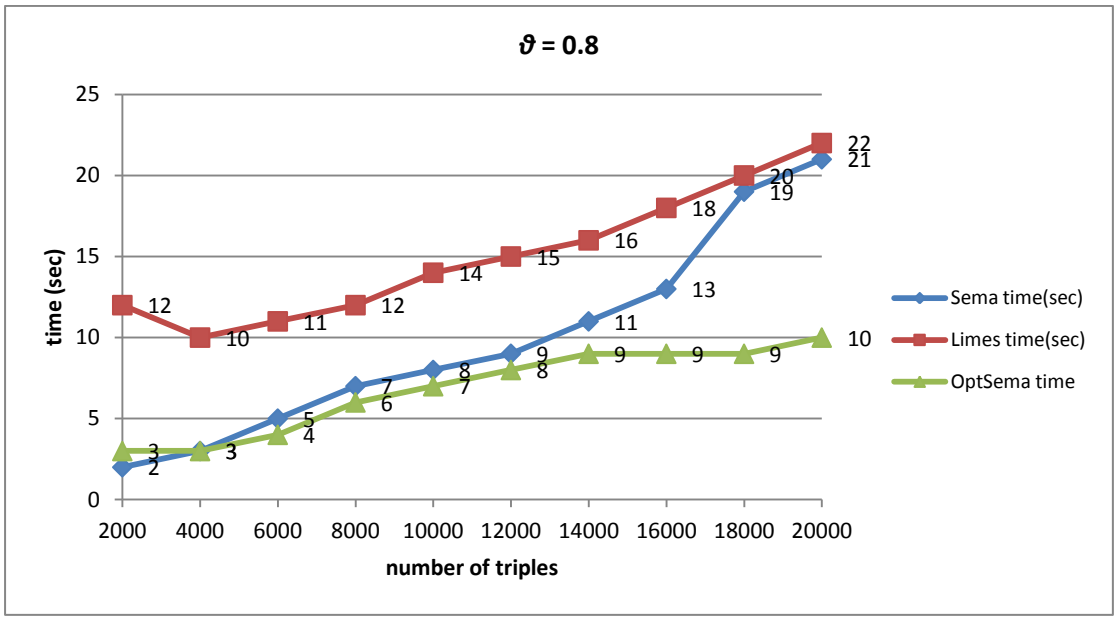
d=0, if threshold $\in$ *(0.5, 1]*

d=1, if threshold $\in$ *(0.33, 0.5]*

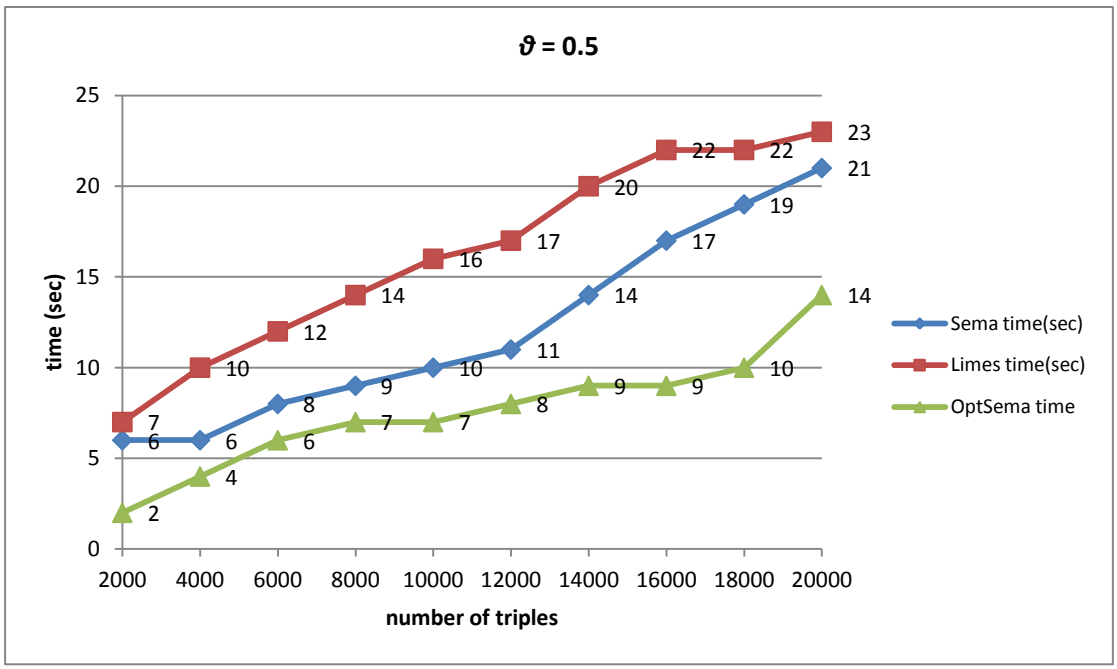
d=2, if threshold $\in$ *(0.25, 0.33]*

d=3, if threshold $\in$ *(0.2, 0.25]*

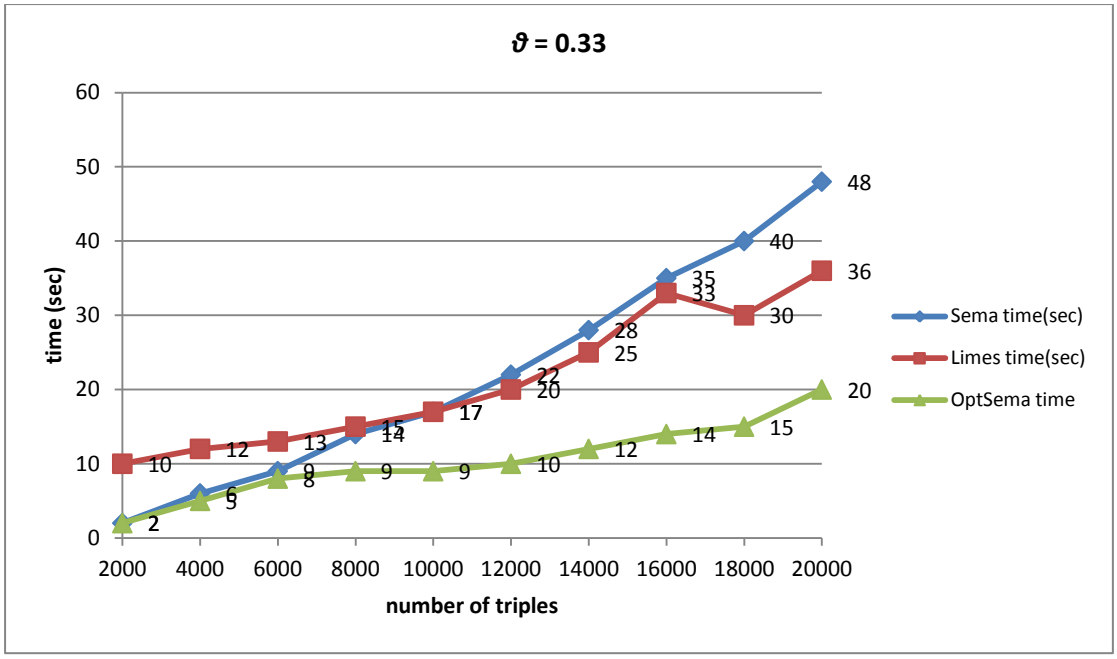
Runtime comparison



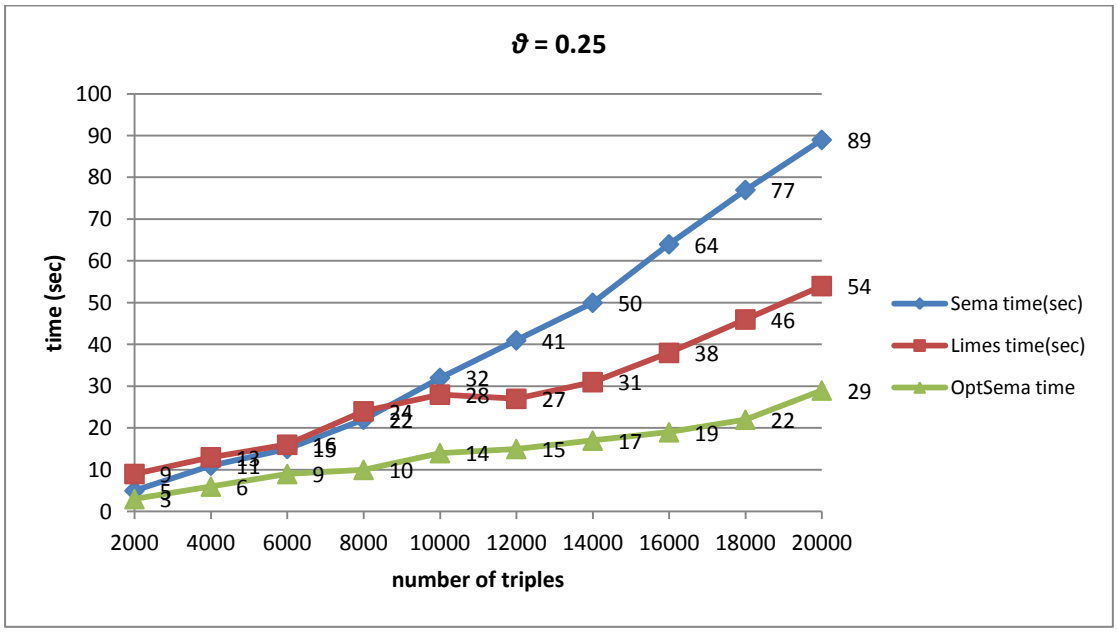
Graph 8 : Runtime comparison for ϑ = 0.8



Graph 9 : Runtime comparison for ϑ = 0.5



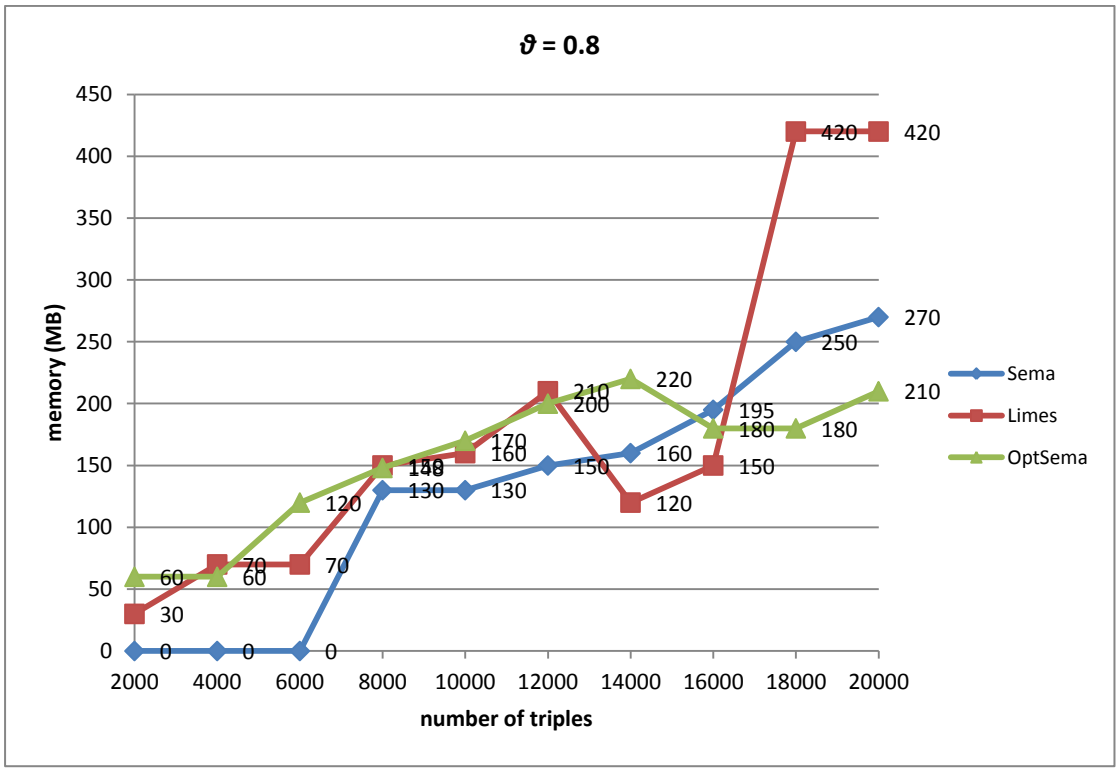
Graph 10 : Runtime comparison for ϑ = 0.33



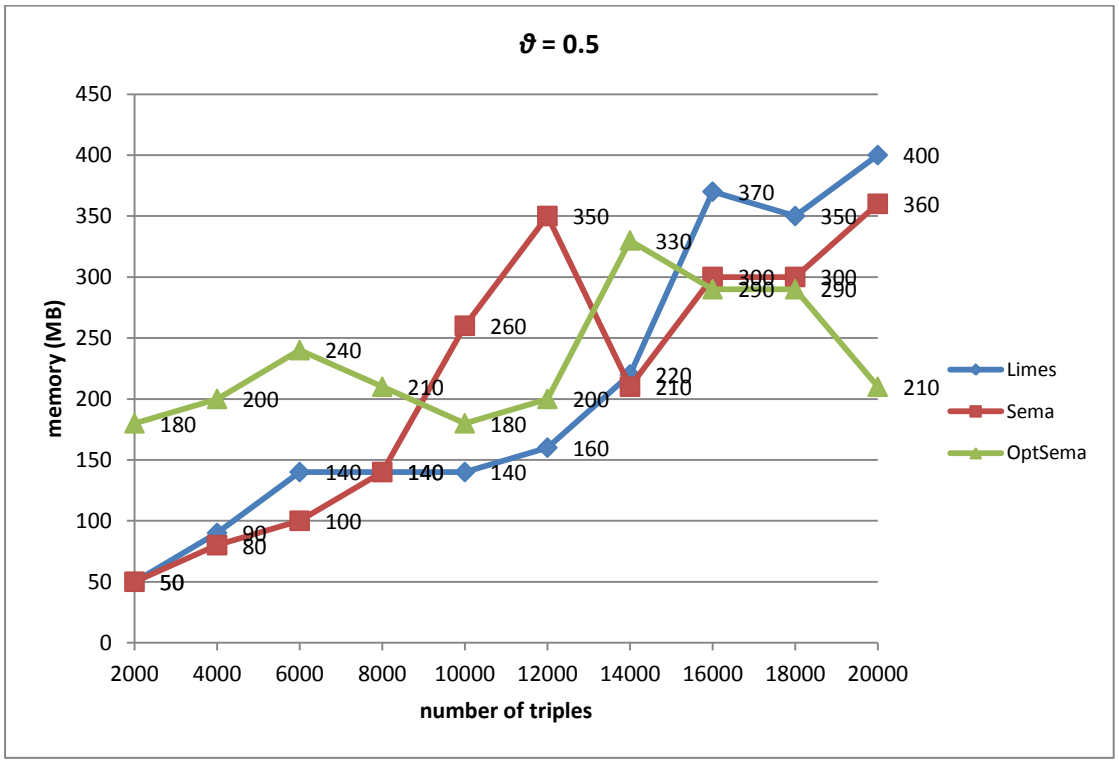
Graph 11 : Runtime comparison for ϑ = 0.25

From the above graphs we can see that Sema has a good performance only for ϑ= 0.8 and 0.5, in other words only for none or one edit operation. In all the other cases LIMES outperforms Sema as we can observe on the graphs as we increase the number of triples the difference expands. On the other hand our optimized version called OptSema has better results in terms of runtime for all the thresholds ϑ. The difference in runtime steadily increases, as we increase the number of triples. This is more obvious in the last experiment where we use 20000 triples. For 2000 triples OptSema finishes the matching task in 3 seconds while LIMES finishes the matching task in 9 seconds and the difference is 6 seconds. For 20000 triples OptSema finishes the matching task in 29 seconds while LIMES finishes the matching task in 54 seconds so the difference expands to 25 seconds.

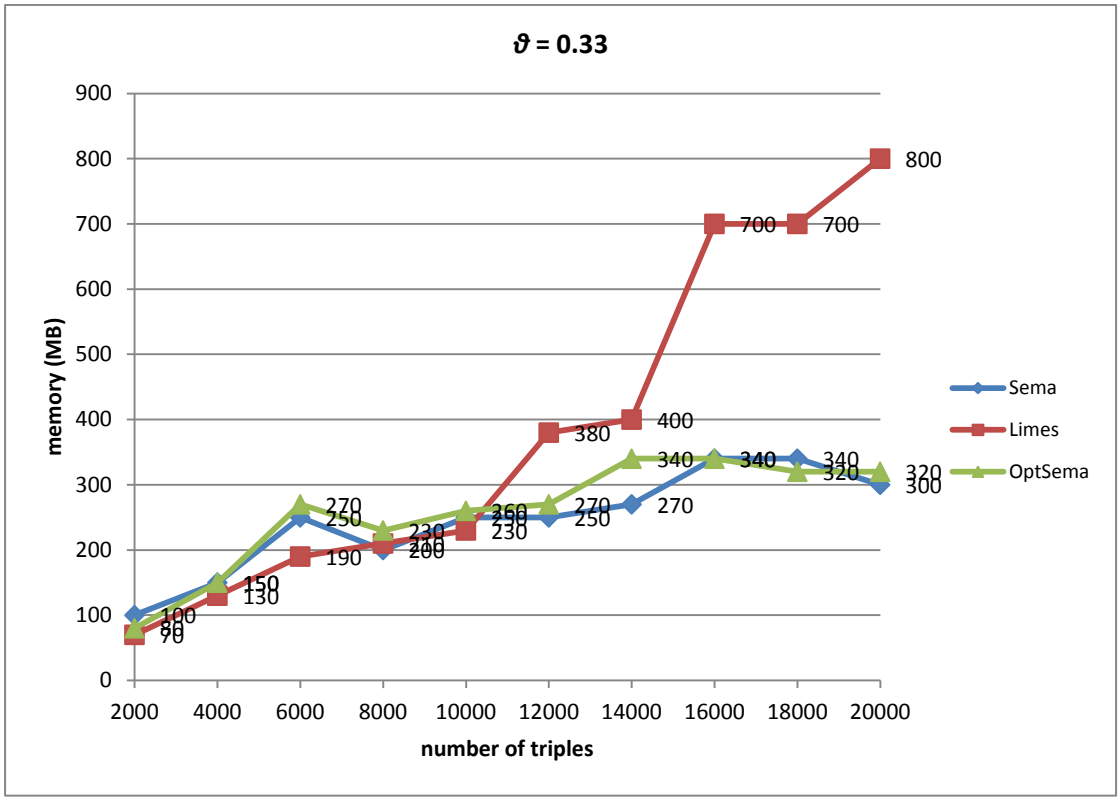
Memory comparison



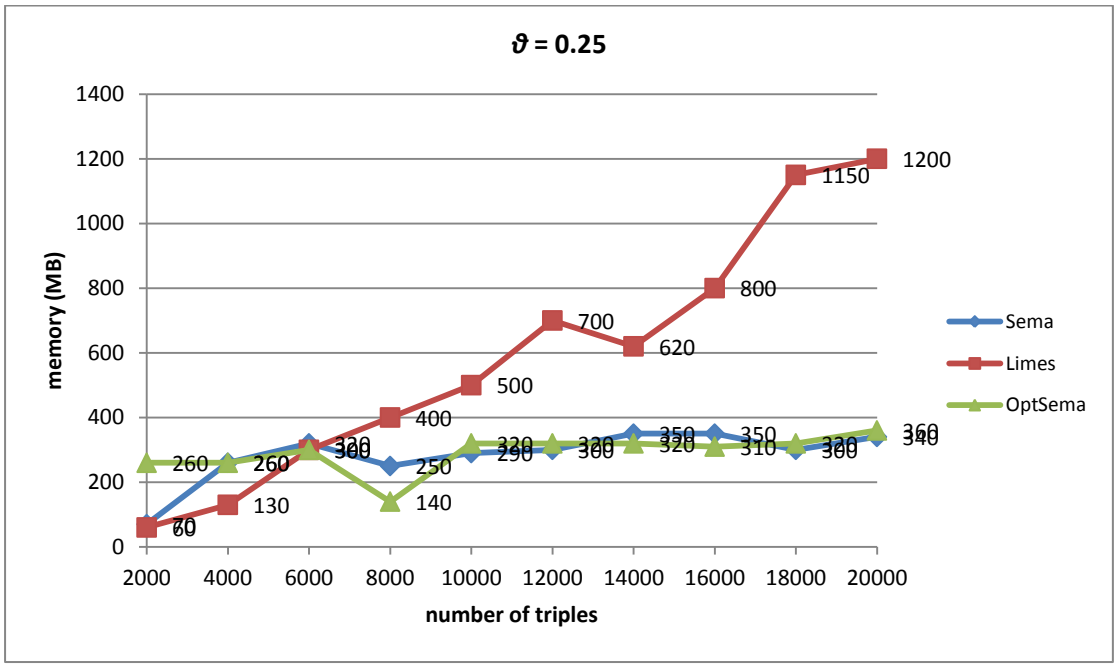
Graph 12 : Memory footprint for ϑ = 0.8



Graph 13: Memory footprint for ϑ = 0.5



Graph 14: Memory footprint for ϑ = 0.33



Graph 15 : Memory footprint for ϑ = 0.25

From the graphs above we can deduce that both our implementations Sema and OptSema have approximately the same behavior in memory usage. From the last two experiments it is clear that both methods achieve better results in memory usage than LIMES.

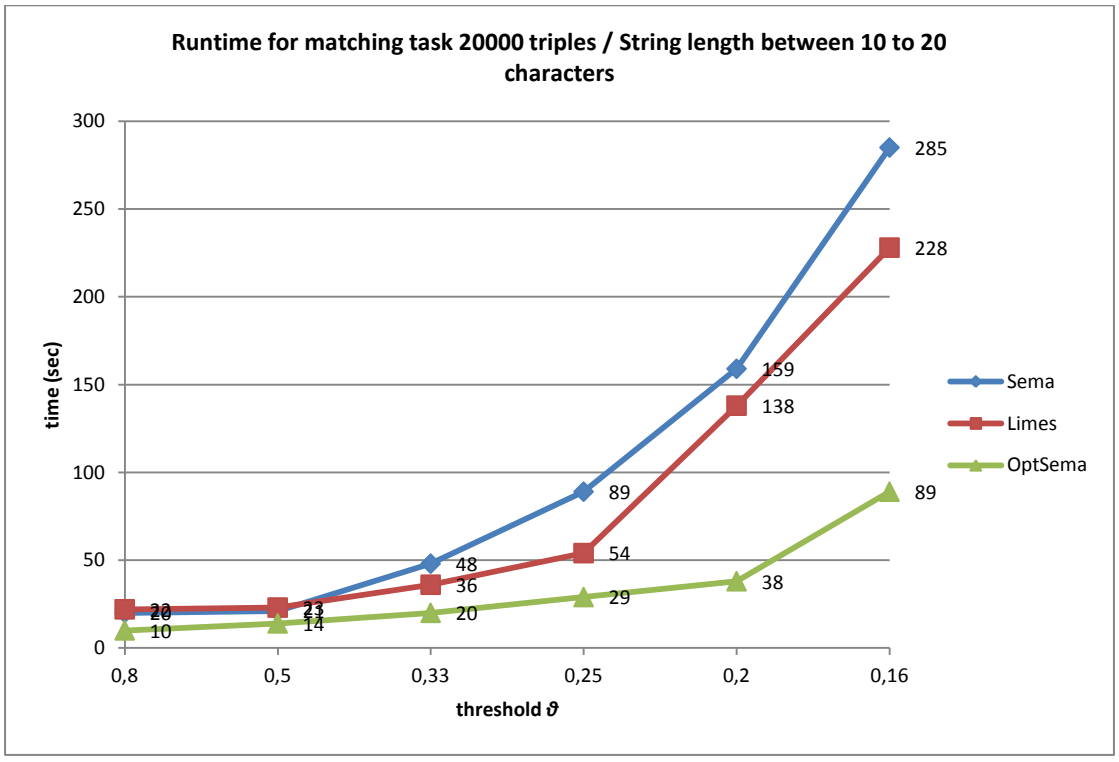
e. Comparison for 20000 strings with 10-20 & 20-30 characters

Comparison for strings between 10 -20 characters

We performed two experiments with two datasets that contained 20000 triples. The first dataset contained strings whose length varied from 10 to 20 characters. The second dataset contained strings whose length varied from 20 to 30 characters. The purpose is to see the performance of these methods for different lengths.

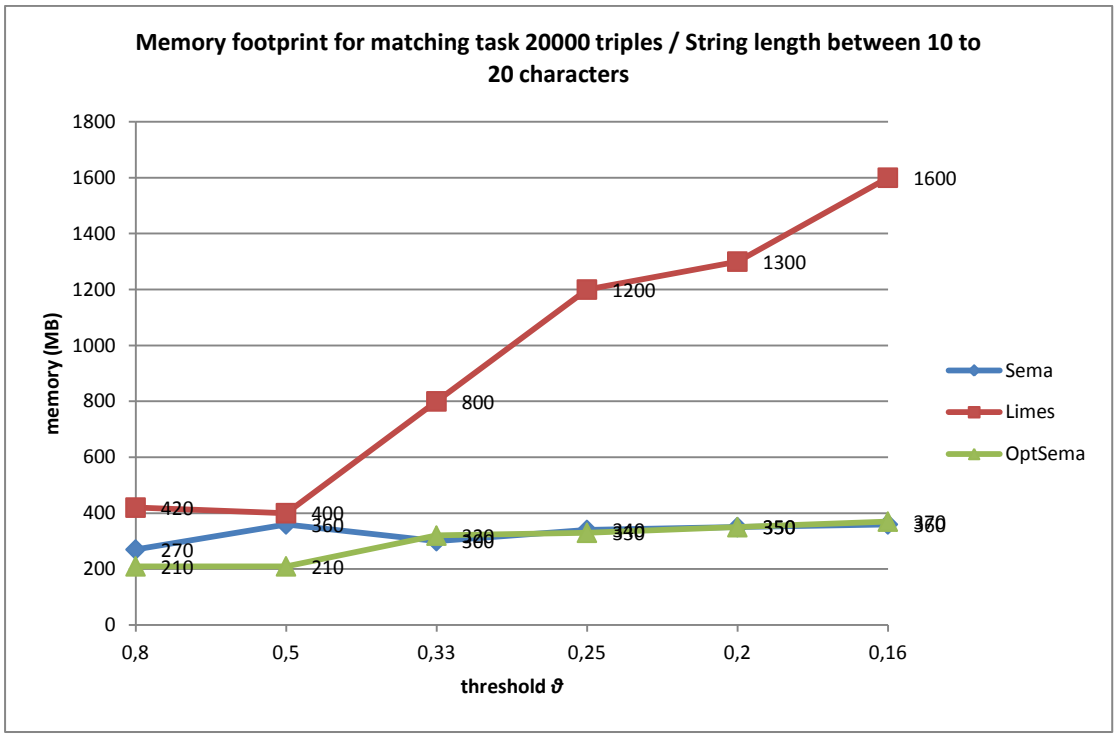
In the first case we have tested LIMES, Sema, OptSema using strings with lengths that vary from 10 to 20 characters. The dataset used contained 20000 triples and the threshold ϑ varies from the value 0,8 to the value 0,16. When ϑ is equal to 0.2 this means that we have 4 edit operations while 0.16 means that we have 5 edit operations. In these two cases we match two strings even if they need 1/5 or 1/4 of their length as edit operations in order to align. The following graphs show the comparison of the three methods in terms of runtime and memory usage.

Runtime comparison



Graph 16 : Runtime comparison from ϑ = 0.8 to ϑ = 0.16

Memory comparison



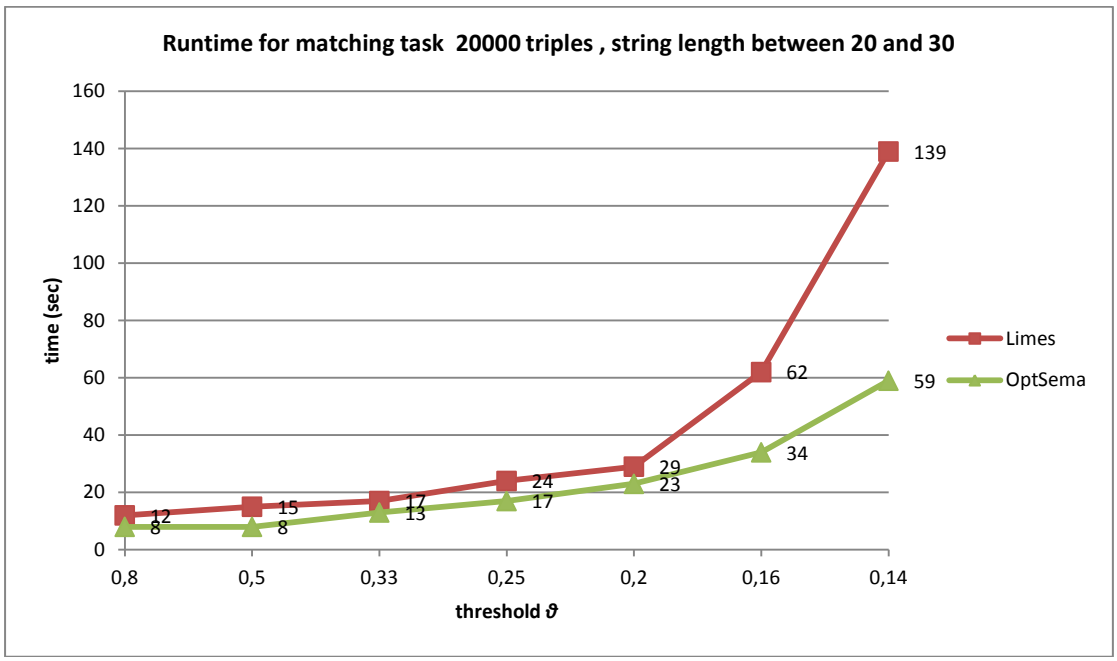
Graph 17 : Memory footprint from ϑ = 0.8 to ϑ =0.16

From the above graphs we can see that OptSema outperforms LIMES both in runtime and memory footprint, but now we have a few more findings. All methods have worst performance as we decrease ϑ but OptSema has a more smooth change in its' behavior, while LIMES has a very abrupt change.

Comparison for strings between 20 -30 characters

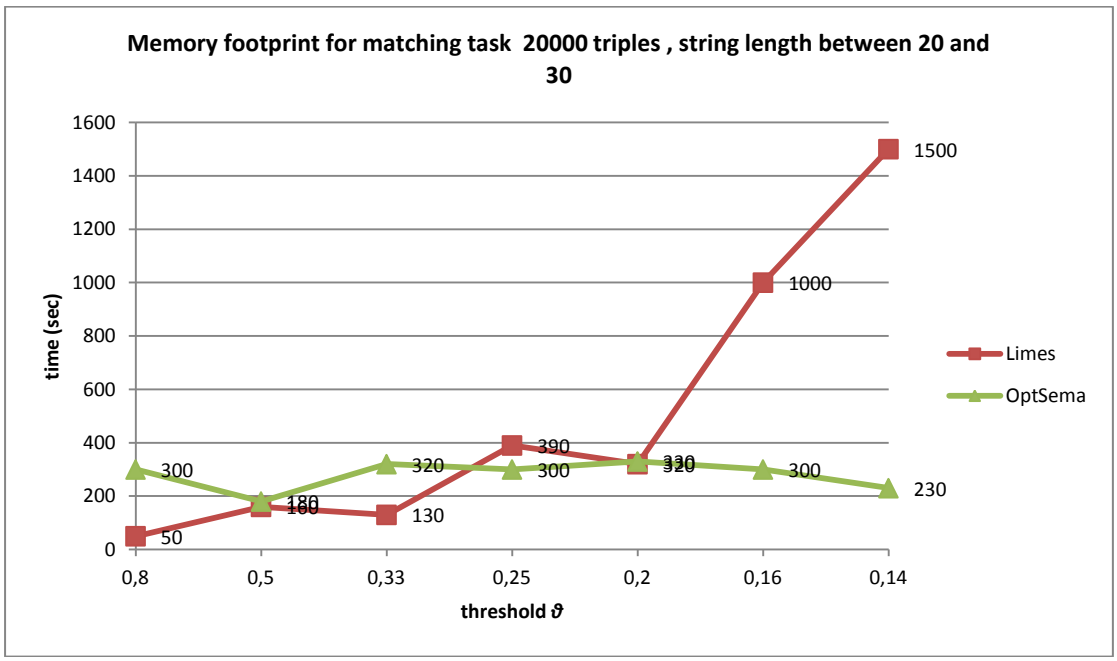
We have tested LIMES, OptSema using strings with lengths that vary from 20 to 30 characters. The dataset used contained 20000 triples and the threshold ϑ varies from the value 0,8 to the value 0,14. We performed this experiment to see the behavior of our best method OptSema in comparison to LIMES in larger strings. The following graphs show the comparison of the methods in terms of runtime and memory usage.

Runtime comparison



Graph 18 : Runtime comparison from ϑ = 0.8 to ϑ = 0.14

Memory comparison



Graph 19 : Memory footprint from ϑ = 0.8 to ϑ =0.14

From the above graphs we can see that OptSema outperforms LIMES both in runtime and memory footprint. We can also observe that even though we have larger strings OptSema performs better than before and LIMES performs much better than before (see Graph 16). This due to fact that the pruning power of our method depends on the length of the segments produced in the blocks of the target dataset. So since we have larger strings this

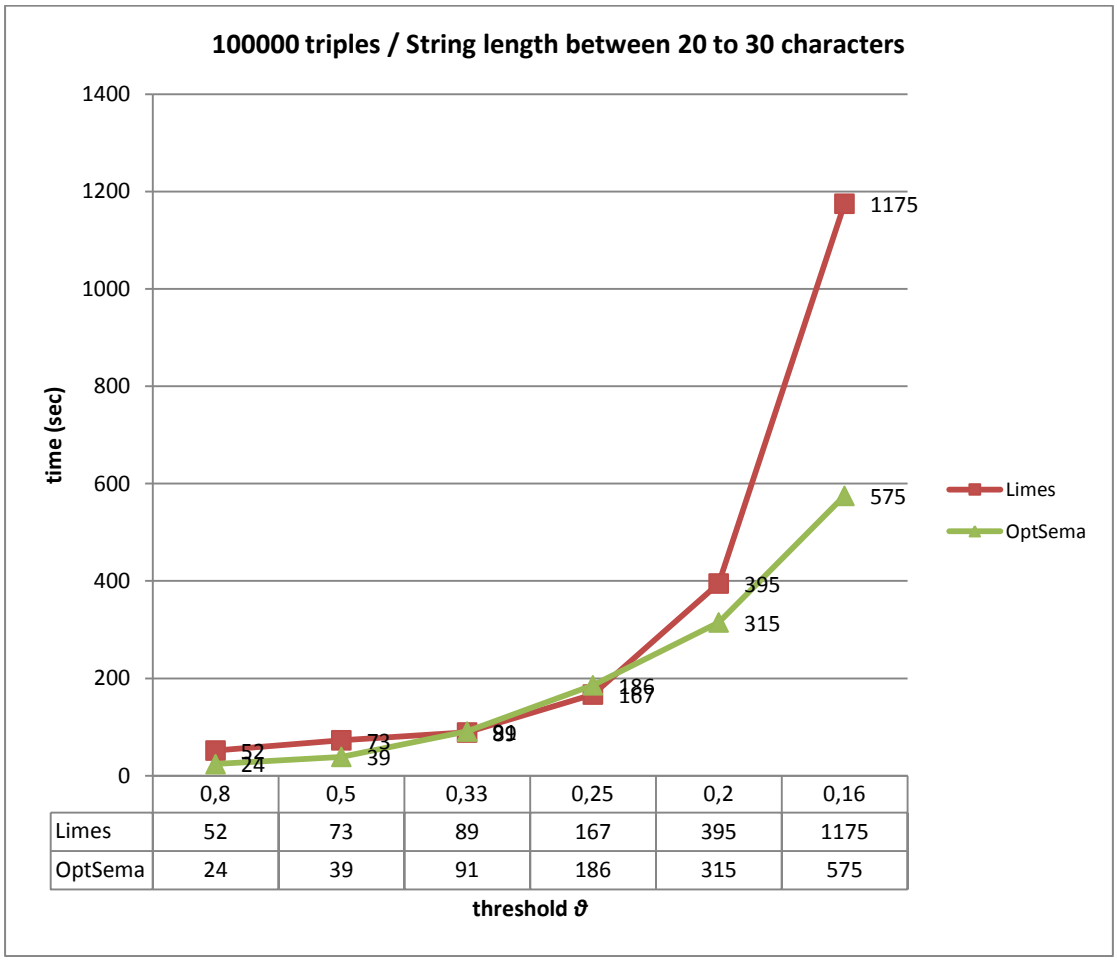
means that we create larger segments that are less likely to be found as a substring of a source string.

f. OptSema vs LIMES on large datasets

100000 triples – Strings vary from 20 to 30 characters

We have tested again Limes, OptSema using strings with lengths that vary from 20 to 30 characters. The dataset used contained 100000 triples and the threshold ϑ varies from the value 0.8 to the value 0.16. The following graphs show the comparison of the two methods in terms of runtime and memory usage.

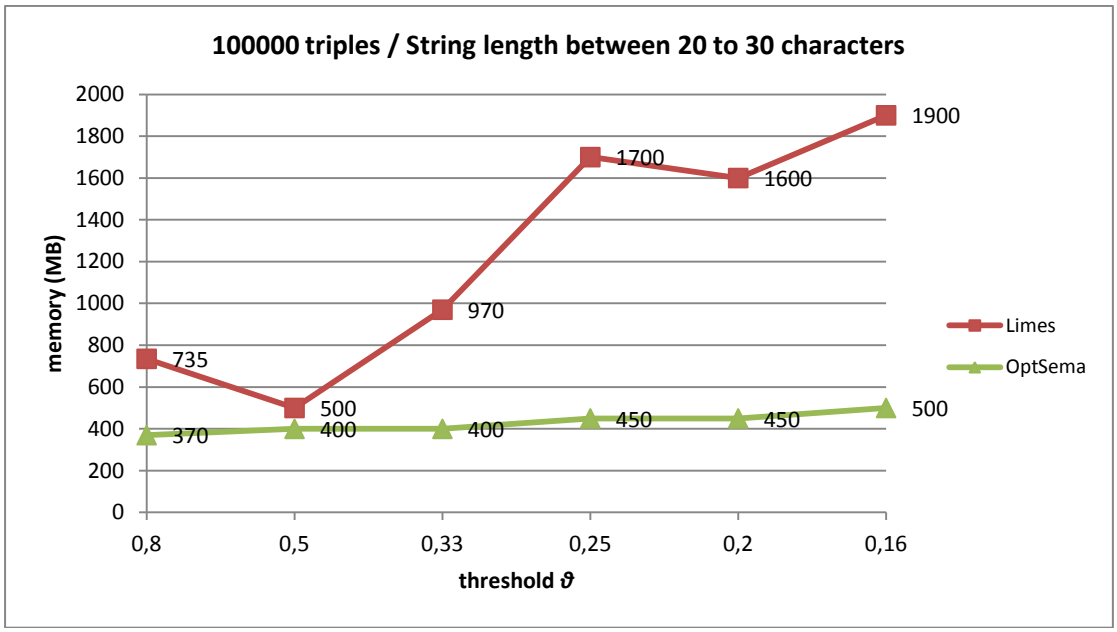
Runtime comparison



	0,8	0,5	0,33	0,25	0,2	0,16
Limes	52	73	89	167	395	1175
OptSema	24	39	91	186	315	575

Graph 20 : Runtime comparison from ϑ = 0.8 to ϑ = 0.16 for 100000 triples

Memory comparison



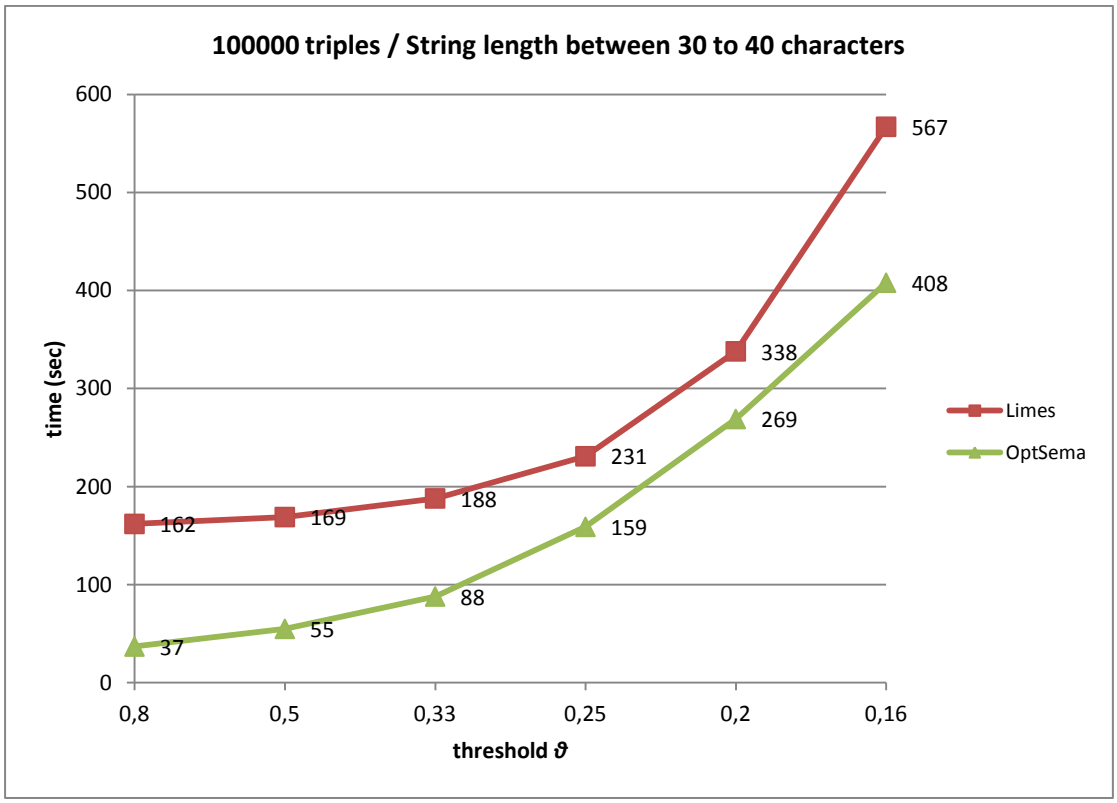
Graph 21 : Memory footprint from ϑ = 0.8 to ϑ =0.16 for 100000 triples

We can see that there are two cases when θ is equal to 0,33 and 0,25 that LIMES has a better runtime. In all the other cases OptSema performs better. From the aspect of memory usage again we can observe that OptSema performs better in all the cases.

100000 triples – Strings vary from 30 to 40 characters

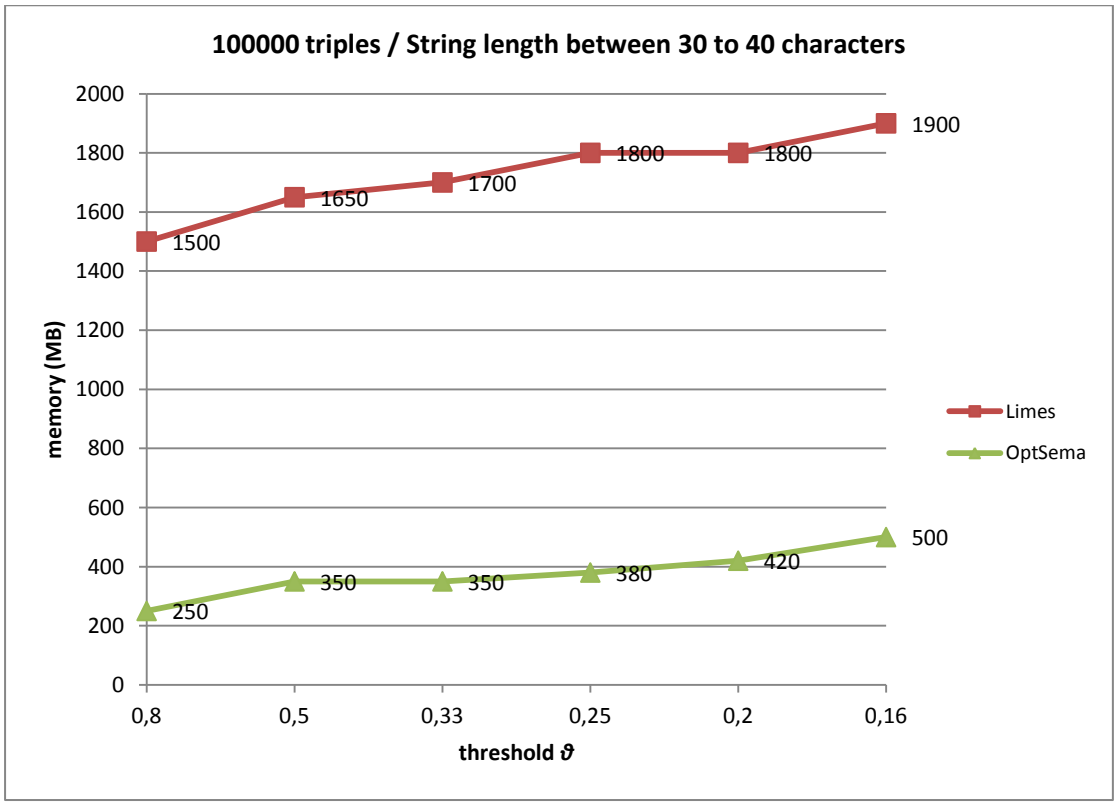
We have tested again LIMES, OptSema using strings with lengths that vary from 30 to 40 characters. The dataset used contained 100000 triples and the threshold ϑ varies from the value 0,8 to the value 0,16. The following graphs show the comparison of the two methods in terms of runtime and memory usage.

Runtime comparison



Graph 22 : Runtime comparison from ϑ = 0.8 to ϑ = 0.16 for 100000 triples

Memory comparison



Graph 23 : Memory footprint from ϑ = 0.8 to ϑ =0.16 for 100000 triples

We can see that in all the cases tested OptSema performs better than LIMES. From the aspect of memory usage again we can observe that OptSema performs better in all the cases. In comparison to the previous experiment we can deduce that our algorithm performs better when the threshold ϑ is small and it represents a small number of edit operations compared to the length of the string. For example, the runtime for ϑ = 0.2 (i.e. 4 edit operations) is 315 seconds for the length 20-30 characters, while in this case the runtime is 269 for the length 30 -40 characters.

11. Conclusion

This work proposed and evaluated a time and space efficient approach for computing links between data sources, exploiting string similarities. The proposed approach builds on a basic indexing method that facilitates pruning dissimilar pairs and verifying candidate pairs, effectively. It proposed a filtering approach that utilizes string lengths and further partitions the target source to blocks of entities according to dissimilarities between strings of equal length. For the verification stage, this work presented methods that aim to reduce the complexity for computing the string metric used to quantify the similarity between two strings. We have created two implementations (Sema, OptSema) and we have compared them to LIMES which is a state-of-the-art framework for link discovery.

We have presented a blocking method that efficiently divides the target dataset T to clusters based on the length of strings and then to blocks by creating exemplars per cluster. We evaluated two methods for the creation of blocks the "First Fit" and the "Best Fit" approach concerning the pruning power of these methods when we apply the Triangle Inequality filter. We showed that "Best Fit" leads to better results. Subsequently, we have presented a method that creates an index per block that contains string segments and the strings that contain this segment. All the above summarizes the methods applied on the target dataset.

We have presented three filters that can be applied in a sequential manner per query: (1) length filter, (2) triangle inequality filter and (3) matching substring filter. We have used these filters per query to produce a small number of candidate strings for verification. We showed that these filters applied produce a small number of candidate strings by testing their pruning power for different thresholds ϑ. We showed the relation between number of exemplars and the pruning power of the Triangle Inequality filter and the Matching Substring filter.

We have presented two implementations of the matching substring filter. The first implementation was used in the version of our algorithm called Sema. The second implementation was used in the version of our algorithm called OptSema. Both implementations lead to the same number of comparisons that have be performed in order to complete a matching task. The OptSema performs better in terms of runtime compared to Sema.

We evaluated OptSema, Sema and LIMES using datasets of different size that contained strings with various lengths. We showed that Sema and OptSema have approximately the same memory footprint. All the experiments have shown that both versions have a better memory footprint than LIMES. In terms of runtime LIMES outperforms Sema. This becomes clearer as we increase the number of triples of the dataset as well as the length of the strings. On the other hand the optimized implementation of our algorithm OptSema outperforms LIMES in terms of runtime. This becomes clearer as we increase the triples of our dataset as well as the length of the strings.

The indexing method that we have introduced made a big difference in the efficiency and runtime of the matching substring filter and made our method competitive to LIMES in terms of runtime. We can deduce this conclusion from the fact that the only difference between Sema and OptSema is the implementation of the matching substring filter.

In terms of memory usage LIMES has proven to be a very memory consuming framework especially in cases where the threshold ϑ was very small and it had to deliver a large number of matches.

Future work

This work could serve as a starting point for the creation of a more complex and delicate framework for link discovery that efficiently performs a more complex matching task. The first thing that could be implemented is to add more string similarity metrics such as trigrams, Jaccard and cosine that could be more appropriate than Levenshtein distance depending on the property that will be used for the matching task. For example Trigrams is a more appropriate string similarity metric for long strings. Another function that would add value to our framework is to be able to combine more than one condition that has to be met in order to match two instances. This combination of conditions could concern more than one property of a data item. Also these conditions could refer to properties that can have all the data types that are used for the creation of Linked Data like numerical values, strings, dates etc.

12. Bibliography

[1] Arasu, A., Ganti, V., & Kaushik, R.(2006). Efficient exact set-similarity joins. *In VLDB*, 918–929.

[2] Bayardo, R. J., Ma, Y. & Srikant, R. (2007). Scaling up all pairs similarity search. *In WWW,* pages 131–140.

[3] Bizer, C.,Jentzsch, A., Isele, R.(2011) Efficient Multidimensional Blocking for Link Discovery without losing Recall. *14th International Workshop on the Web and Databases (WebDB 2011),* June 12, 2011 - Athens, Greece

[4] Bocek, T., Burkhard, S., Hunt, E.(2007). Fast Similarity Search in Large Dictionaries. *Technical Report ifi-2007.02*, Department of Informatics, University of Zurich.

[5] Feng, J., Wang, J., & Li, G. (2011). Trie-join: a trie-based method for efficient string similarity joins. *The VLDB Journal,21*(4), 437-461. doi:10.1007/s00778-011-0252-8

[6] Hillner, S., & Ngomo, A. N. (2011). Parallelizing LIMES for large-scale link discovery. *Proceedings of the 7th International Conference on Semantic Systems - I-Semantics '11*. doi:10.1145/2063518.2063520

[7] Jiang, Y., Li, G., Feng, J., & Li, W. (2014). String similarity joins. *Proceedings of the VLDB Endowment,7*(8), 625-636. doi:10.14778/2732296.2732299

[8] Levenshtein V. I. (1966) Binary codes capable of correcting deletions, insertions, and reversals. *Technical Report 8*.Retrieved January 30, 2017,
from https://nymity.ch/sybilhunting/pdf/Levenshtein1966a.pdf

[9] Li, G., Deng, D., Wang, J., & Feng, J. (2011). Pass-join. *Proceedings of the VLDB Endowment,5*(3), 253-264. doi:10.14778/2078331.2078340

[10] LIMES — Agile Knowledge Engineering and Semantic Web (AKSW). (n.d.). Retrieved January 30, 2017, from http://aksw.org/Projects/LIMES.html

[11] Linked Data. (n.d.). Retrieved January 30, 2017, from

https://www.w3.org/DesignIssues/LinkedData.html

[12] Nentwig, M., Hartung, M., Ngomo, A. N., & Rahm, E. (2016). A survey of current Link Discovery frameworks. *Semantic Web,8*(3), 419-436. doi:10.3233/sw-150210

[13] Ngonga Ngomo, A.-C. & Auer, Sö. (2011). LIMES - A Time-Efficient Approach for Large-Scale Link Discovery on the Web of Data. *Proceedings of IJCAI*, .

[14] Ngonga Ngomo, A.-C. & Lyko, K. (2012). EAGLE: Efficient Active Learning of Link Specifications using Genetic Programming. *Proceedings of ESWC*, .

[15] Ngonga Ngomo, A.-C. (2012). On Link Discovery using a Hybrid Approach. *Journal on Data Semantics*, 1, 203 -- 217.

[16] Ngonga Ngomo, A.-C. (2014). HELIOS -- Execution Optimization for Link Discovery. *Proceedings of ISWC*, .

[17] OWL. (n.d.). Retrieved January 30, 2017, from https://www.w3.org/OWL/

[18] Papadakis, G., Ioannou, E., Palpanas, T., Niederee, C., & Nejdl, W. (2013). A Blocking Framework for Entity Resolution in Highly Heterogeneous Information Spaces. *IEEE Transactions on Knowledge and Data Engineering,25*(12), 2665-2682. doi:10.1109/tkde.2012.150

[19] Qin, J., Wang, W., Lu, Y., Xiao, C., & Lin, X. (2011). Efficient exact edit similarity query processing with the asymmetric signature scheme. *Proceedings of the 2011 international conference on Management of data - SIGMOD '11*. doi:10.1145/1989323.1989431

[20] RDF. (n.d.). Retrieved January 30, 2017, from https://www.w3.org/RDF/

[21] Sherif, M. A. & Ngonga Ngomo, A.-C. (2015). An Optimization Approach for Load Balancing in Parallel Link Discovery. *SEMANTiCS 2015*, .

[22] Silk. (n.d.). Retrieved January 30, 2017, from http://silkframework.org/

[23] Sun, Y., Ma, L., Shuang, W.: A comparative evaluation of string similarity metrics for ontology alignement. J. Inf. Comput. Sci. **12**(3), 957–964 (2015)

[24] The Linking Open Data cloud diagram. (n.d.). Retrieved January 30, 2017, from http://lod-cloud.net/

[25] Volz, J., Bizer, C., Gaedke, M., & Kobilarov, G. (2009). Discovering and Maintaining Links on the Web of Data. *Lecture Notes in Computer Science The Semantic Web - ISWC 2009,* 650-665. doi:10.1007/978-3-642-04930-9_41

[26] Volz, J., Bizer, C., Gaedke, M., & Kobilarov, G. (2009). Silk – A Link Discovery Framework for the Web of Data . *2nd Workshop about Linked Data on the Web (LDOW2009)*, Madrid, Spain, April 2009.

[27] Wang, J., Li, G., & Feng, J. (2012). Can we beat the prefix filtering? *Proceedings of the 2012 international conference on Management of Data - SIGMOD '12*. doi:10.1145/2213836.2213847

[28] Wang, W., Qin, J., Xiao, C., Lin, X., & Shen, H. T. (2013). VChunkJoin: An Efficient Algorithm for Edit Similarity Joins. *IEEE Transactions on Knowledge and Data Engineering,25*(8), 1916-1929. doi:10.1109/tkde.2012.79

[29] Winkler, W. (1999). The state of record linkage and current research problems. *Technical report, Statistical Research Division, U.S. Bureau of the Census*, 1999.

[30] Winkler, W. (2006). Overview of record linkage and current research directions. *Technical report, Bureau of the Census - Research Report Series*, 2006.

[31] Xiao, C., Wang, W., & Lin, X. (2008). Ed-Join. *Proceedings of the VLDB Endowment,1*(1), 933-944. doi:10.14778/1453856.1453957

[32] Xiao, C., Wang, W., Lin, X., & Yu, J. X. (2008). Efficient similarity joins for near duplicate detection. *Proceeding of the 17th international conference on World Wide Web - WWW '08*. doi:10.1145/1367497.1367516