國立臺灣大學電機資訊學院資訊工程學系

碩士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

可有效回覆字串相似度搜尋之演算法

A Query-Efficient Algorithm for String Similarity Search

黃宥勝

You-Sheng Huang

指導教授：趙坤茂博士

Advisor: Kun-Mao Chao, Ph.D.

中華民國 104 年 6 月

June, 2015

國立臺灣大學碩士學位論文

口試委員會審定書

可有效回覆字串相似度搜尋之演算法

A Query-Efficient Algorithm for String-Similarity Search

本論文係黃宥勝君（學號 R02922064）在國立臺灣大學資訊工程學系完成之碩士學位論文，於民國 104 年 6 月 8 日承下列考試委員審查通過及口試及格，特此證明

口試委員：

趙坤茂
（指導教授）

黃耀廷　　王弘倫

系主任　趙坤茂

Acknowledgments

To begin with, I would like to express my gratitude to my advisor, Prof. Kun-Mao Chao. His guidance plays an important role in me finishing this thesis. In addition to the academic help, the encouragement from him is an important factor for me in achieving many goals. I also want to thank my examination commissioners, Prof. Hung-Lung Wang and Prof. Yao-Ting Huang, who provided me with precious advice on my thesis. Furthermore, I appreciate all assistance I have received from partners in ACB Lab. Above all, I thank my family for all their support.

摘要

編輯距離 (edit distance) 是一個廣泛地被用於測量字串之間相似程度的度量，而字串相似度搜尋 (string similarity search) 則要找出在特定的字串集合中和給予的查詢字串 (query string) 相似的字串。以編輯距離為測量依據的字串相似度搜尋，被大量地應用在如數據清理 (database cleaning) 、錯誤檢查更正 (Error detection and correction) 與資料擷取 (data retrieval) 等領域。目前此類問題的解決方法大多先將可排除的字串過濾掉後，再對剩下的字串進行檢驗。然而，這些方法在排除字串的過程中，幾乎全部都採用相同的過濾規則，使得效率隨著字串集合的改變而下降。為了克服這個問題，我們提出一個整合不同過濾方法的資料結構 (data structure) 讓字串的過濾維持穩定的效率。我們並提出對應的演算法解決兩個主要的字串相似度搜尋問題：範圍查詢 (range query) 與前 k 個查詢 (top-k query) 。實驗結果顯示當門檻值小於一定的程度時，我們的方法在範圍查詢上具有優異的性能表現。

關鍵字：編輯距離、字串相似度搜尋、範圍查詢、前 k 個查詢

Abstract

Edit distance, a measure determining the similarity between two strings, is a criterion that has been used widely. String similarity search finds strings in a dataset that are similar to a given query string. Edit-distance based string similarity search is exploited in many fields, e.g., database cleaning, error detection and correction and data retrieval. Most approaches toward string similarity search resort to filtering out as many strings in datasets as possible and verifying the remaining strings. However, these approaches use only one filtering method throughout the whole procedure, which makes the power of the method fluctuates during the whole procedure. To overcome this problem, we propose a data structure integrating different filtering methods and adopting a more efficient one on each phase. We also give corresponding algorithms for two important queries of string similarity search, range query and top-k query. Experimental results show that our approach is competitive for range query when thresholds are small enough.

Keywords: edit distance, string similarity search, range query, top-k query

Contents

口試委員會審定書 **i**

Acknowledgments **i**

摘要 **ii**

Abstract **iii**

Contents **iv**

List of Figures **vi**

List of Tables **vii**

1 Introduction **1**

1.1 Related Work 2

1.1.1 Previous Work 2

1.1.2 Verifying Algorithm 3

1.1.3 Pruning Technique 4

1.2 Our Work 5

2 Preliminaries **6**

2.1 Preliminary Knowledge 7

2.1.1 Edit Distance 7

2.1.2 Computations of Edit Distance 7

2.2 Problem Definition 8

2.2.1 Range Query 8

2.2.2 Top-k Query 8

3 Complex-Tree Based Solution 9

3.1 Filtering Methods 9

3.1.1 String Length 10

3.1.2 Letter Count 10

3.1.3 Reference String 10

3.2 Index Construction 11

3.2.1 String Length 11

3.2.2 Letter Count 12

3.2.3 Reference String 14

3.3 Query Algorithms 14

3.3.1 Range Query 14

3.3.2 Top-k Query 17

4 Experimental Evaluation 18

4.1 Setup 18

4.2 Space Consumption 19

4.3 Comparison with Other Approaches 19

4.3.1 Range Query 19

4.3.2 Top-k Query 20

5 Conclusions and Future Work 22

5.1 Conclusions 22

5.2 Future Work 22

List of Figures

3.1 Structure example. 13

4.1 Average response time for range query. 20

4.2 Average response time for top-k query. 20

List of Tables

2.1 Notation. 6

3.1 Running example. 9

3.2 Mapping example. 13

3.3 Reference string choosing example. 14

3.4 Boundary restriction example with $\theta = 2$. 16

4.1 Dataset statistics. 18

4.2 Index construction time in second. 19

4.3 Space consumption for index construction in MB. 19

Chapter 1

Introduction

String similarity search is an important operation used in a wide variety of applications, e.g., error checking, data cleaning, data integration and pattern recognition. It discovers strings in datasets that are similar to query strings according to a given distance measure. For example, in error checking, which finds possible typos in a given document, string similarity search finds words not in a given dictionary and recommends similar words determined by edit distance.

Various distance measures are used in different fields based on their characteristics [1, 2], and edit distance is one of the most widely adopted distance measures in relevant topics. It is therefore also employed in our work.

Because of the need of a both time and space efficient solution for answering queries with small thresholds, which is the case in many applications (e.g., error checking, data cleaning), we are interested in how to respond to queries efficiently using reasonable space.

Due to the costly complexity, $O(|s|\theta)$, of verifying whether two strings with length s have edit distance smaller than or equal to a given threshold θ, our approach, like most other existing ones [3,5,7,8,11–14], tries to filter out as many strings in datasets as possible to minimize the verification needed.

In this thesis, a query-efficient algorithm using acceptable space is proposed. Given that various filtering methods outperform others in certain datasets [5], different filtering methods are integrated in our approach and adopted on different phases respectively. Experiments on real datasets show that our approach outperforms others when the thresholds are small enough.

1.1 Related Work

Section 1.1.1 introduces recent studies related to the topic of string similarity search. The algorithms and techniques exploited in our work are explained in the following subsections.

1.1.1 Previous Work

Edit distance, which is also referred to as Levenshtein distance, is one of the research topics that have been studied for a long period of time. A number of algorithms for computing edit distance have been proposed over the past few decades, and there is usually a trade-off between time and space complexities. At the moment, to the best of our knowledge, the fastest algorithm has time complexity $O(|s|^2/\log|s|)$ [16]; on the other hand, the most space-efficient algorithm has space complexity $O(\max\{|s_1|, |s_2|\})$ [17].

The research of string similarity search is another topic that has gotten a lot of attention. In contrast with edit distance, various algorithms for efficiently solving string similarity search problems are still being proposed at the moment. Most of the algorithms do their best to filter out as many strings in the datasets as possible, so that the verifications needed, which consume most of the time, are minimized. The filtering methods used can be roughly categorized into following two groups.

N-gram Based Inverted Index [3–10]: In this approach, a number of inverted lists are maintained for answering strings in datasets sharing certain number of n-grams with the query. Obviously, the performance of methods using this approach mainly depends on the algorithms they use to merge the inverted lists. An efficient merging algorithm is proposed in [5]. Although it is efficient with a small threshold, its efficiency degrades when query strings or strings in datasets are short. Also, it suffers from severe space overhead.

B$^+$-tree [14,15]: It is first advanced by Zhang et al. [14]. An order of strings in datasets is defined beforehand, which enables the indexes to be stored in a B$^+$-tree. The B$^+$-tree is then used by different algorithms to answer queries. Another efficient string order and algorithm is proposed by Lu et al. [15] These approaches can usually have good performance; however, it suffers from search overhead when inappropriate string orders are chosen, which vary from one dataset to another.

1.1.2 Verifying Algorithm

Verification, which determines whether two strings have edit distance smaller than or equal to a given threshold, is an intensely used operation in string similarity search. Instead of computing the actual edit distance between two strings, a more efficient approach, Algorithm VerifyED, is proposed in [14].

Algorithm 1 VerifyED (string s_1, string s_2, distance threshold θ)

```
if $||s_1| - |s_2|| > \theta$ then
    return FALSE
end if
Construct a table $T$ of 2 rows and $|s2| + 1$ columns
for $j = 1$ to $\min\{|s_2| + 1, 1 + \theta\}$ do
    $T[1][j] = j - 1$
end for
Set $m = \theta + 1$
for $i = 2$ to $|s_1| + 1$ do
    for $j$= $\min\{1, i - \theta\}$ to $\min\{|s_2| + 1, i + \theta\}$ do
        $d_1 = (j < i + \theta)\ ?\ T[1][j] + 1\ :\ \theta + 1$
        $d_2 = (j > 1)\ ?\ T[2][j - 1] + 1\ :\ \theta + 1$
        $d_3 = (j > 1)\ ?\ T[1][j - 1] + (s_1[i - 1] = s_2[j - 1])\ ?\ 0\ :\ 1)\ :\ \theta + 1$
        $T[2][j] = \min\{d1, d2, d3\}$
        $m = \min\{m, T[2][j]\}$
    end for
    if $m > \theta$ then
        return FALSE
    end if
    for $j = 0$ to $|s_2| + 1$ do
        $T[1][j] = T[2][j]$
    end for
end for
return TRUE
```

The idea of Algorithm 1 is that only the entries of the dynamic programming table on the diagonal with offset no larger than the threshold need testing. It is easy to observe that entries not tested leads to a larger edit distance than θ; therefore, they are simply skipped.

By this algorithm, a verification operation can be finished in $O(|s|\theta)$, where s is the shorter string being verified.

1.1.3 Pruning Technique

A partition based pruning solution is proposed in [15]. Strings in datasets are split into several partitions. Each partition P_i is characterized by a representative reference string o_i. The minimum edit distance $P_i.l$ and maximum edit distance $P_i.u$ between o_i and any strings in P_i is maintained. Formally, $P_i.l = \min\{ed(s, o_i)|s \in P_i\}$, $P_i.u = \max\{ed(s, o_i)|s \in P_i\}$, where $ed(s, o_i)$ is the edit distance between s and o_i. In the rest of this these, we use $ed(s_1, s_2)$ to denote the edit distance between s_1 and s_2. When a query is issued, partitions are pruned according to the following rules:

Given a range query with threshold θ and query string q, a partition P is a prunable partition if and only if one of the following conditions holds: (1) $ed(o, q) - P.u > \theta$; or (2) $P.l - ed(o, q) > \theta$.

Pruning Rule 1. *If partition P is a prunable partition, then $\forall s \in P, ed(s, q) > \theta$. Thus, strings that lie in P do not have to be verified and can be pruned directly.*

Given a range query with threshold θ and query string q, a partition P is a candidate partition if and only if one of the following conditions holds: (1) $P.l \leq ed(o, q) \leq P.u$; or (2) $P.l - \theta \leq ed(o, q) \leq P.l$; or (3) $P.u \leq ed(o, q) \leq P.u + \theta$.

Pruning Rule 2. *Suppose P is a candidate partition. Then, $\forall s \in P$, s needs to be verified if and only if*

$$lb \leq ed(o, s) \leq ub$$

where $lb = \max\{ed(o, q) - \theta, P.l\}$, $ub = \min\{ed(o, q) + \theta, P.u\}$. We refer to range $[lb, ub]$ as **candidate region.**

Given a range query with threshold θ and query string q, a partition P is a selectable partition if and only if the following condition holds: $ed(o, q) + P.u \leq \theta$.

Pruning Rule 3. *Suppose that partition P is a selectable partition. Then, $\forall s \in P$, $ed(s, q) \leq \theta$, and s is reported as a result. We refer to range $[P.l, P.u]$ as* **selectable region.**

If a partition is prunable or selectable, no further verification on the strings in the partition is required. If the partition is a candidate partition, then only strings that lie in the candidate region need to be verified.

1.2 Our Work

In this thesis, we introduce a new approach to string similarity search. Our approach is based on the fact that different filtering methods outperform others in certain datasets. Rougher filtering methods are used first, which greatly reduces the candidate strings in few steps. The more precise filtering methods are then adopted to diminish the candidate strings further.

In this thesis, we make following contributions:

- We propose an indexing structure that integrates several filtering methods.
- We propose algorithms for string similarity search based on the structure.
- A trade-off between query efficiency and memory consumption is provided, which makes it feasible to deal with tremendous datasets.
- We conduct experiments to evaluate the performance of our approach and compare it with others.

The rest of this thesis is structured as follows. Chapter 2 gives the background and formal definitions of string similarity search problem. Chapter 3 gives details of the approach we propose. Chapter 4 presents the experiments we conducted, and Chapter 5 concludes this thesis.

Chapter 2

Preliminaries

In this chapter, some preliminary knowledge and problem definitions are presented. In the rest of this thesis, we assume Σ a finite alphabet. For string s, the length of it is denoted by $|s|$, and the i_{th} letter is denoted by $s[i]$, where $s[i] \in \Sigma$ and $1 \leq i \leq |s|$. The notation used in the rest of this thesis are summarized in Table 2.1.

Table 2.1: Notation.

Notation	**Explanation**
Σ	the alphabet
s	a string consists of letters in Σ
$\|s\|$	the length of string s
$s[i]$	the i_{th} letter of string s
$s[i,j]$	the substring of s from the i_{th} letter to the j_{th} letter
q	a query string
θ	a threshold of edit distance
$ed(q,s)$	the edit distance between q and s
$c(s,l)$	the number of occurrences of letter l in s
$h(n)$	the height of the tree containing node n
$d(n)$	the level of node n
$v(n)$	the value of node n.
$l(n)$	the letter implied by nodes with level equal to n.
$p(n,k)$	the ancestor of n with level equal to k.

2.1 Preliminary Knowledge

2.1.1 Edit Distance

There are three edit operations on string s: insertion, deletion and substitution. An insertion operation inserts a character $x \in \Sigma$ into s, which forms a new string $s' = s[1, i-1]$ $x\ s[i, |s|]$ with length $|s|+1$. A deletion operation removes a letter $s[i]$ from s, which forms a new string $s' = s[1, i-1]\ s[i+1, |s|]$ with length $|s|-1$. A substitution operation replaces a letter $s[i]$ with a character $x \in \Sigma$, which forms a new string $s' = s[1, i-1]\ x\ s[i+1, |s|]$ with length $|s|$. The edit distance is defined as follow:

Definition 2.1 (Edit Distance)**.** *Given two strings s_i and s_j, the edit distance between s_i and s_j, denoted by $ed(s_i, s_j)$, is defined as the minimum number of edit operations needed to transform s_i to s_j.*

2.1.2 Computations of Edit Distance

Besides Verification, of which Algorithm VerifyED gives an efficient solution, the computation of exact edit distances is also needed in our approach for both constructing indexing structures and answering queries. It can be computed using dynamic programming. Algorithm 2 gives details of the computation using dynamic programming.

Algorithm 2 ED (string s_1, string s_2, int len_s_1, int len_s_2)

```
if $len\_s_1 == 0$ then
    return $len\_s_2$
end if
if $len\_s_2 == 0$ then
    return $len\_s_1$
end if
if $s_1[len\_s_1 - 1] == s_2[len\_s_2 - 1]$ then
    $cost = 0$
else
    $cost = 1$
end if
return minimum( ED($s_1$, $s_2$, $len\_s_1 - 1$, $len\_s_2$)+1,
                ED($s_1$, $s_2$, $len\_s_1$, $len\_s_2 - 1$)+1,
                ED($s_1$, $s_2$, $len\_s_1 - 1$, $len\_s_2 - 1$)+$cost$ )
```

2.2 Problem Definition

We can then give the definitions of string similarity queries based on edit distance. Section 2.2.1 and 2.2.2 give formal definitions of range query and top-k query, which are the most frequently considered queries respectively.

2.2.1 Range Query

Range query is one of the most important query types of string similarity. In many cases, it returns more meaningful results than top-k query, which sometimes returns results that deviate from query strings too much.

Definition 2.2 (Range Query)**.** *Given a query string q and a string set $S = \{s_1, s_2, ..., s_{|S|}\}$, a range query returns a subset S' of S with edit distance to q no larger than θ, i.e., $S' = \{s_i \in S | ed(s_i, q) \leq \theta\}$.*

2.2.2 Top-k Query

Top-k query, another important query type of string similarity, returns k strings that are the most similar to query strings.

Definition 2.3 (Top-k Query)**.** *Given a query string q and a string set $S = \{s_1, s_2, ..., s_{|S|}\}$, a Top-$k$ query returns a subset S' of k strings in S with edit distance to q no larger than that of any other string in $S - S'$.*

Chapter 3

Complex-Tree Based Solution

The idea of our approach is to adopt different filtering methods on each phase, which optimizes the power of pruning strings. We make efforts on choosing the filtering methods used on each phase and how to integrate them into our structure. Section 3.1 gives the properties of filtering methods used. We then demonstrate how to build a complex-tree integrating these filtering methods in Section 3.2. Finally, we present efficient algorithms for answering both range and top-k queries using complex-tree in Section 3.3.

3.1 Filtering Methods

We present three filtering methods in this section, which are **string length**, **letter count** and **reference string** respectively.

Table 3.1: Running example.

String	**String Content**	$\|s\|$	g_1	g_2	g_3
q	Li Zongyo	9	2	3	4
s_1	Li Zongyong	11	2	5	4
s_2	Li Zou	6	2	1	3
s_3	Liu Zongtian	12	4	5	3
s_4	Liu Zongyu	10	4	3	3
s_5	Xi Zongyue	10	4	3	3
s_6	Xi Zoleyue	10	4	1	5
s_7	Xing Zouxl	10	4	3	3

3.1.1 String Length

String length is one of the most intuitive ways to filer out strings, Property 3.1 gives an accurate way to do it.

Property 3.1 (String Length Pruning)**.** *Given a range query with query string q and threshold θ, strings in the dataset with length larger than $|q|+\theta$ or less than $|q|-\theta$ are prunable.*

Consider the sample query string and dataset shown in Table 3.1. Given that the threshold is 2, both string s_2 and s_3 are prunable due to the excess of $|q| + \theta$, 11, or the insufficiency of $|q| - \theta$, 7, in length.

3.1.2 Letter Count

Letter count is another filtering method that is intuitive but efficient. Property 3.2 gives a simple solution; however, it is refined further when we integrate it into our structure.

Property 3.2 (Letter Count Pruning)**.** *Given a range query with query string q and threshold θ. For string s, let b_l be the sum of every difference between $c(s, a_i)$ and $c(q, a_i)$, where a_i is each letter in s satisfying that $c(s, a_i) > c(q, a_i)$, i.e., $b_l = \sum c(s, a_i) - c(q, a_i)|a_i \in s, c(s, a_i) > c(q, a_i)$, and let b_r be the sum of every difference between $c(q, a_j)$ and $c(s, a_j)$, where a_j is each letter in q satisfying that $c(q, a_j) > c(s, a_j)$, i.e., $b_r = \sum c(q, a_j) - c(s, a_j)|a_j \in q, c(q, a_j) > c(s, a_j)$. If $b_l > \theta$ or $b_r > \theta$, s is prunable.*

Consider the sample query string and dataset shown in Table 3.1. Given that the threshold is 2, string s_2 with $b_r = 4$, s_3 with $b_l = 5$, s_5 with $b_l = 3$, s_6 with $b_l = 5$ and s_7 with $b_l = 4$ are prunable.

3.1.3 Reference String

Similar to the technique mentioned in Section 1.1.3, we can filter strings out based on their edit distance to another specific reference string.

Property 3.3 (Reference String Pruning)**.** *Given a reference string r and a range query with query string q and threshold θ, if string s satisfies that $|ed(s, r) - ed(r, q)| > \theta$, s is prunable.*

Consider the sample query string and dataset shown in Table 3.1. Given that the threshold is 2 and the reference string is s_4, both string s_4 and s_7 are prunable.

3.2 Index Construction

Once the filtering methods used are determined, we can build complex-trees based on these filtering methods. Unlike general search trees, a complex-tree has different criteria for traversing on different levels, which are string length, letter count and the edit distance to a specific reference string respectively. Algorithm 3 and Algorithm 4 describe the details of construction. To choose the desired reference string, Algorithm 3 should be followed by Algorithm 4.

Algorithm 3 Construction (string s, Complex-tree node N, alphabet size t)

```
if $d(N) == 0$ then
    if $N$ has no child $n_c$ such that $v(n_c) = |s|$ then
        Add node $n_c$ with value $= |s|$ into $N$'s children.
    end if
    Construction($s$, $n_c$)
end if
if $1 \leq d(N) \leq t - 1$ then
    if $N$ has no child $n_c$ such that $v(n_c) = c(s, l(d(N)))$ then
        Add node $n_c$ with value $= c(s, l(d(N)))$ into $N$'s children.
    end if
    Construction($s$, $n_c$)
end if
if $d(N) == t$ then
    Add node $n_c$ with value $= s$ into $N$'s children.
end if
```

3.2.1 String Length

Strings in datasets are first categorized by their length. That is, for a complex-tree, if node n_c satisfies that $d(n_c) = 1$, the strings indexed by descendants of $d(n_c)$ have length equal to $v(n_c)$.

Algorithm 4 ChoosingReference (Complex-tree node N, alphabet size t)

```
if $d(N)! = t$ then
    for each $n_c \in$ children of $N$ do
        ChoosingReference($n_c$)
    end for
else
    Find the child of $N$, $n_c$, that $v(n_c)$ maximizes the standard deviation of the edit distance between it and all other siblings.
    for each $n_d \in$ children of $N$ do
        if $n_c$ has no child $n_e$ such that $v(n_e) = ed(n_c, n_d)$ then
            Add node $n_e$ with value $= ed(n_c, n_d)$ into $n_c$'s children.
        end if
        Add a node with value $= n_d$ into $n_e$'s children.
        if $n_d! = n_c$ then
            Delete node $n_d$.
        end if
    end for
end if
```

3.2.2 Letter Count

Strings are then categorized by each amount of different letters they contains. That is, for a complex-tree, if node n_c satisfies that $2 \leq d(n_c) \leq h(n) - 3$, letter $l(d(n_c))$ appears $v(n_c)$ times in each string indexed by descendants of n_c.

We may, intuitively, enumerate all letters used in datasets when constructing complex-trees. However, it suffers from severe memory consumption, which is too expensive for many machines. To ease this situation, we use a hash function to map the alphabet into a smaller universe, which is variable in size to fit in different memory capacity.

Consider the sample dataset shown in Table 3.1 and the mapping function shown in Table 3.2. Given that $l(2) = g_1$, $l(3) = g_2$ and $l(4) = g_3$, because s_1 consists of two letters mapped to g_1, five letters mapped to g_2 and five letters mapped to g_3, the ancestors of the node indexing s_1 on level 2, 3 and 4 should have values 2, 5 and 5 respectively.

However, there is no need to record the letter count of the last group, which is inferable by deducting all letter counts of other groups from the string length. As we can see in the structure example shown in Figure 3.1, which is constructed according to the dataset shown in Table 3.1 and the mapping function shown in Table 3.2, the letter count of g_3, which can be computed by deducting the values of corresponding nodes on level 2 and

Table 3.2: Mapping example.

Group	Letter
g_1	i
	t
	u
	x
	y
	X
g_2	a
	g
	n
	(space)
g_3	e
	l
	o
	z
	L
	Z

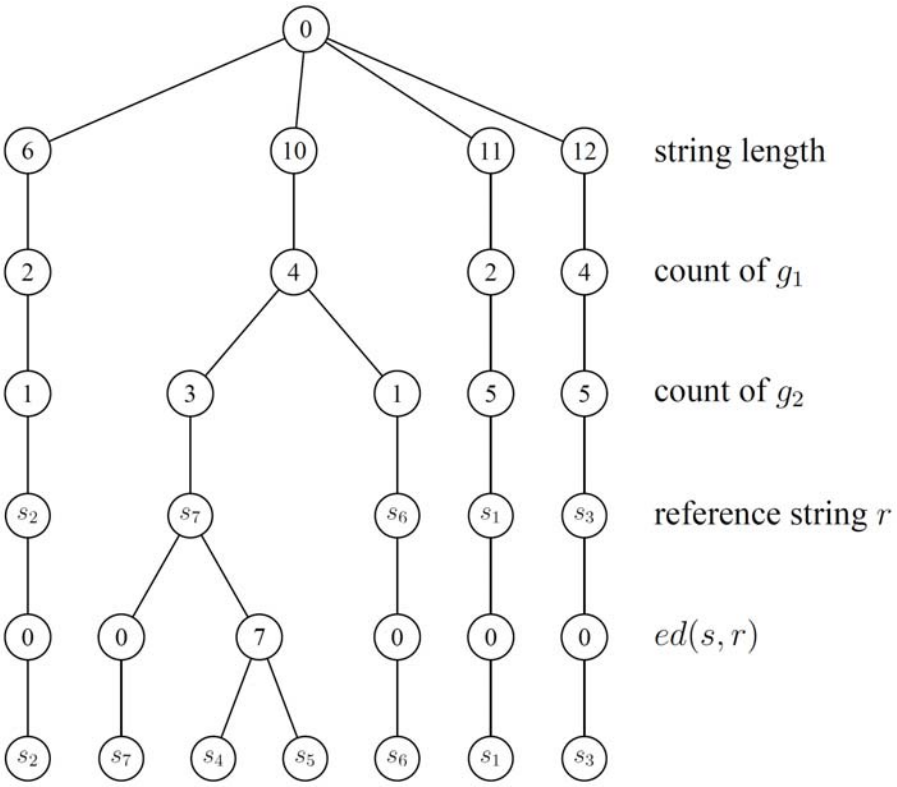

Figure 3.1: Structure example.

level 3 from the value of corresponding node on level 1, is not recorded.

3.2.3 Reference String

Table 3.3: Reference string choosing example.

String	$ed(x, s_4)$	$ed(x, s_5)$	$ed(x, s_7)$	**standard deviation**
s_4	0	3	7	2.87
s_5	3	0	7	2.87
s_7	7	7	0	3.30

After above categorizations are finished, we then choose a string in each set as reference string. The string that makes the edit distances between it and all other strings have the largest standard deviation is chosen.

Consider the strings computed in Table 3.3. The edit distance between each string and the others are computed first, and the standard deviation of them are then calculated. The string making the largest standard deviation, which is s_7 in the example, is chosen as the reference string.

Once the reference string is decided, all strings are then categorized by the edit distance between it and the reference string.

As we can see in Figure 3.1, once s_7 is chosen as the reference string, s_4 and s_5 are then indexed by its child with value 7. Also, s_7 is indexed by its child with value 0.

3.3 Query Algorithms

Once the structure is constructed, we can efficiently respond to queries with the structure.

3.3.1 Range Query

Algorithm 5 gives the details of how to answer range queries with a complex-tree. As the procedure of building the index, every query is processed with three phases, which are string length, letter count and reference string respectively.

```
Algorithm 5 RangeQuery (string $q$, threshold $\theta$, Complex-tree node $N$, int $b_l$, int $b_r$)
  if $d(N) == 0$ then
      for each $n_c \in$ children of $N$ do
          if $|q| - \theta \leq v(n_c) \leq |q| + \theta$ then
              RangeQuery($q$, $\theta$, $n_c$, $\min\{\theta, \theta - (v(n_c) - |q|)\}$, $\min\{\theta, \theta - (|q| - v(n_c))\}$)
          end if
      end for
  end if
  if $1 \leq d(N) \leq h(N) - 3$ then
      for each $n_c \in$ children of $N$ do
          if $c(q, l(d(N))) - b_l \leq v(n_c) \leq c(q, l(d(N))) + b_r$ then
              RangeQuery($q$, $\theta$, $n_c$, $\min\{b_l, b_l - (c(q, l(d(n))) - v(n_c))\}$,
                                              $\min\{b_r, b_r - (v(n_c) - c(q, l(d(n))))\}$)
          end if
      end for
  end if
  if $d(N) == h(n) - 2$ then
      for each $n_c \in$ children of $N$ do
          if $ED(q, v(N), |q|, |v(n)|) - \theta \leq v(n_c) \leq ED(q, v(N), |q|, |v(n)|) + \theta$ then
              for each $n_d \in$ children of $n_c$ do
                  if VerifyED($q, v(n_d), \theta$) then
                      Add $v(n_d)$ into query result
                  end if
              end for
          end if
      end for
  end if
```

For query string q, we first traverse the subtrees with strings indexed having length between $|q| - \theta$ and $|q| + \theta$ based on property 3.1.

However, we can restrict the number of subtree traversed further by denoting the left and right boundary of subtrees to traverse in each iteration.

Property 3.4 (Boundary Restriction)**.** *Given a range query with query string* q *and threshold* θ*, if string* s *is a result string,* s *lacks at most* $\min\{\theta, \theta - (|s| - |q|)\}$ *letters in* q *and has at most* $\min\{\theta, \theta - (|q| - |s|)\}$ *additional letters besides those in* q*.*

Given the query with query string q = Lee Lie and θ = 2. Without loss of generality, consider the letter count of letter 'e', Table 3.4 lists all possible combination of $|s|$ and $c(s,\text{e})$. We can find that there is no result string s containing less than $\min\{2, 2-(|s|-7)\}$ e or more than $\min\{2, 2-(7-|s|)\}$ e.

The algorithm then determines the subtrees to traverse based on each letter count of

Table 3.4: Boundary restriction example with $\theta = 2$.

$\|s\|$	$c(s$,e$)$	s
7	**3**	**Lee Lie** (query string)
5	1	Le Li, L Lie
	2	Lee L, Le Le
	3	Lee e, ee Le, ...
6	1	Le Lia, Le Lib, ...
	2	Lee La, Lee Lb, ...
	3	Lee Le, Lee Ae, ...
	4	Lee ee, ee Lee, ...
7	1	Lea Lia, Lae Lia, ...
	2	Lee Laa, Lee Lab, ...
	3	Lee Lie, Lee Lae, ...
	4	Lee Lee, Lee aee, ...
	5	Lee eee, LeeeLee, ...
8	2	Lee Liaa, Lee Liab, ...
	3	Lee Liea, Lee Laea, ...
	4	Lee Leea, Lee Leea, ...
	5	Lee Leee, LeeeLiee, ...
9	3	Lee Lieaa, Lee Liaea, ...
	4	Lee Lieea, Lee Lieae, ...
	5	Lee Leeea, Leee Liee, ...

q and the parameter b_l and b_r, which denotes the left bound and right bound respectively. Every children n_c of the processed node N satisfying that $c(q, l(d(N))) - b_l \leq v(n_c) \leq c(q, l(d(N))) + b_r$ is traversed. For each iteration, if any quota of b_l or b_r is used, it is deducted in the following iteration.

We then traverse the subtrees indexing strings having edit distance to reference strings between $ed(q, v(N)) - \theta$ and $ed(q, v(N)) + \theta$. All strings indexed by nodes in these subtrees are considered candidate strings.

Finally, for each candidate strings, we use Algorithm 1 to verify whether it is a result string or not. If it is, we add it into a result set.

3.3.2 Top-k Query

Algorithm 6 TopkQuery (string q, int k, Complex-tree node N)

```
nonrepeating set $R = \emptyset$
threshold $\theta = 0$
while $|R| < k$ do
    RangeQuery($q$, $\theta$, $N$, $\theta$, $\theta$)
    add the result into $R$
    $\theta = \theta + 1$
end while
return the $k$ strings in $R$ inserted first
```

Algorithm 6 gives the details of answering Top-k Queries. To answer top-k queries, we make a series of range query and increase the threshold gradually. In each iteration, we add the result into a non repeating set R to make sure that result strings having smaller edit distance to the query string are added first.

Chapter 4

Experimental Evaluation

In this chapter, we give the results of experiments we have conducted. Both range and top-k queries are evaluated.

4.1 Setup

In the experiments, real datasets are used. We use the data of *Author* from DBLP[1]. The data, which may become larger with time, is drawn on January 24, 2015. For each query, we average the response time for 100 query strings, which are generated by keeping random sampling until a set with difference between the average and variance of it and the dataset is less than 1. Table 4.1 shows the information of the dataset and the query string we generate.

Table 4.1: Dataset statistics.

Dataset	**Cardinality**	**Min.**	**Max.**	**Avg.**	**Var.**
Author	1529933	3	78	15	5
Query	100	6	41	15	5

In the experiments, we compare our approach, denoted by **CT**, to B^{ed}-tree [14] with two different string orders it proposes, denoted by **BD** and **BGC** respectively.

[1] http://dblp.uni-trier.de/db/

All programs are compiled by G++ 4.9.2. Experiments are conducted on a machine running Linux and equipped with Xeon Processor E5-2620 and 128 GB main memory.

The index construction time spent by the methods is listed in Table 4.2. The index should be constructed only once for each dataset.

Table 4.2: Index construction time in second.

	CT	**BD**	**BGC**
Author	19	46	21

4.2 Space Consumption

We first test the space used for building the indexing structure. The space used on every layer are listed in Table 4.3.

Table 4.3: Space consumption for index construction in MB.

String Length	**Letter Count**	**Reference String**	**Total**
0.00002	168.51	78.86	247.38

4.3 Comparison with Other Approaches

In this section, we compare the results of our approach with others. Range and top-k queries are evaluated in subsections 4.3.1 and 4.3.2 respectively.

4.3.1 Range Query

In Figure 4.1, we compare the average response time for range queries with different thresholds on the author data from DBLP. The results show that our approach outperforms others significantly for small thresholds and is more efficient when thresholds are smaller than 5. Complex-tree is more efficient than other approaches for small threshold because its pruning power does not degrade with the change of level. When threshold is

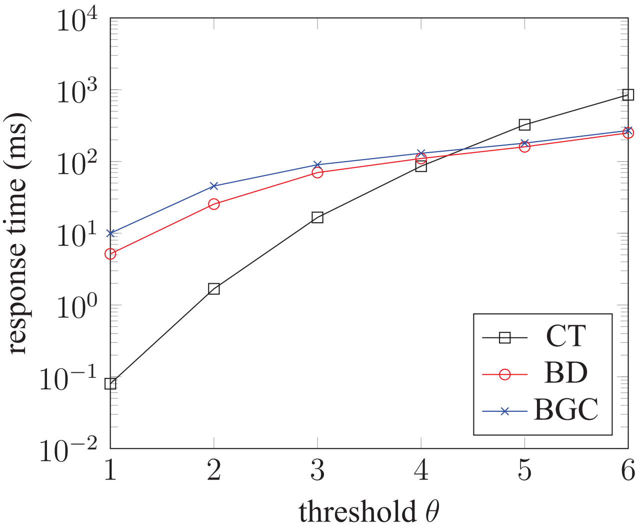

Figure 4.1: Average response time for range query.

small (1 and 2), Complex-tree can almost response queries immediately. Although the strings to verify in Complex-tree expand faster than that in other approaches do, it keeps its advantage for a long period.

4.3.2 Top-k Query

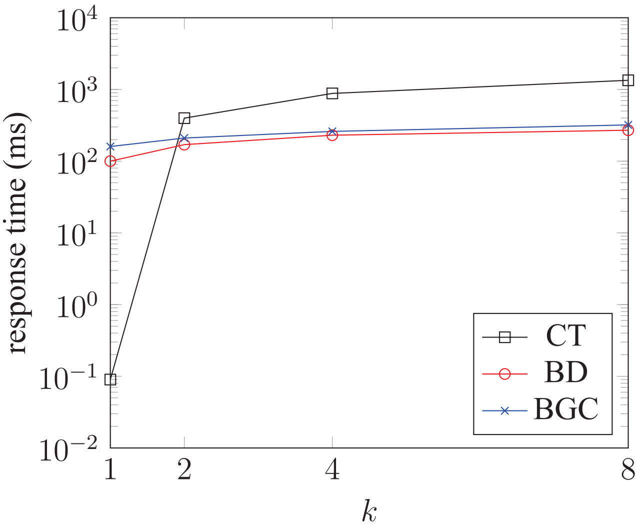

Figure 4.2: Average response time for top-k query.

Figure 4.2 shows the average response time for top-k queries with different thresholds on the author data from DBLP. From the result, we can find that Complex-tree bests other approaches only when k is less than 2. However, we argue that it is mainly caused by the result strings that have large edit distance to query strings, which are usually meaningless in many cases.

Chapter 5

Conclusions and Future Work

In this chapter, we conclude what we achieve in this thesis in Section 5.1. We also suggest some directions for future work, which make the approach more powerful and practical.

5.1 Conclusions

In this thesis, we propose an indexing structure integrating different filtering methods and algorithms using this structure to answer range queries and top-k queries. The experimental results show that our approach outperforms others when the thresholds are small enough.

5.2 Future Work

One direction to enhance our approach would be to integrate more powerful filtering methods into the indexing structure. It could be either compatible to the present structure or more powerful than an old filtering method used.

Another progress to make is to generalize the hash function mapping the alphabet into another universe. The hash function in our implement supports only ASCII now. Making it support Unicode enables the algorithm to deal with more datasets and be more practical.

Bibliography

[1] A. Chandel, O. Hassanzadeh, N. Koudas, M. Sadoghi, and D. Srivastava. Benchmarking declarative approximate selection predicates. In SIGMOD, pp. 353-364, 2007.

[2] S. Chaudhuri, B.-C. Chen, V. Ganti, and R. Kaushik. Exampledriven design of efficient record matching queries, In VLDB, pp. 327-338, 2007.

[3] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In VLDB, pp. 491-500, 2001.

[4] C. Li, B. Wang, and X. Yang. Vgram: Improving performance of approximate queries on string collections using variable-length grams. In VLDB, pp. 303-314, 2007.

[5] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In ICDE, pp. 257-266, 2008.

[6] R. Vernicaand and C. Li. Efficient top-k algorithms for fuzzy search in string collections. In KEYS ’ 09: Proceedings of the First International Workshop on Keyword Search on Structured Data, pp. 9-14, 2009.

[7] J. Qin, W. Wang, Y. Lu, C. Xiao, and X. Lin. Efficient exact edit similarity query processing with the asymmetric signature scheme. In SIGMOD Conference, pp. 1033-1044, 2011.

[8] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In SIGMOD Conference, pp. 85-96, 2012.

[9] A. Behm, S. Ji, C. Li, and J. Lu. Space-constrained gram-based indexing for efficient approximate string search. In ICDE, pp. 604-615, 2009.

[10] A. Behm, C. Li, and M. J. Carey. Answering approximate string queries on large data sets using external memory. In ICDE, pp. 888-899, 2011.

[11] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In ICDE, pp. 5, 2006.

[12] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In VLDB, pp. 918-929, 2006.

[13] G. Li, D. Deng, J. Wang, and J. Feng. Pass-join: A partition-based method for similarity joins. In PVLDB, vol. 5, no. 3, pp. 253-264, 2011.

[14] Z. Zhang, M. Hadjieleftheriou, B. C. Ooi, and D. Srivastava. Bedtree: an all-purpose index structure for string similarity search based on edit distance. In SIGMOD, pp. 915-926, 2010.

[15] W. Lu, X. Du, M. Hadjieleftheriou, and B. C. Ooi. Efficiently supporting edit distance based string similarity search using B^+-trees. In IEEE Transactions on Knowledge and Data Engineering, vol.26, no.12, pp.2983-2996, 2014.

[16] W. J. Masek and M. Paterson. A faster algorithm computing string edit distances. In Journal of Computer and System Sciences, 20(1):18-31, 1980.

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to Algorithms. In MIT Press and McGraw-Hill, 2001.