

# FPI: A Novel Indexing Method Using Frequent Patterns for Approximate String Searches

Mitsuki Kimura  
The University of Tokyo  
2-1-2 Hitotsubashi,  
Chiyoda-ku, Tokyo, JAPAN  
mick@nii.ac.jp

Atsuhiro Takasu  
National Institute of Informatics  
2-1-2 Hitotsubashi,  
Chiyoda-ku, Tokyo, JAPAN  
takasu@nii.ac.jp

Jun Adachi  
National Institute of Informatics  
2-1-2 Hitotsubashi,  
Chiyoda-ku, Tokyo, JAPAN  
adachi@nii.ac.jp

## ABSTRACT

Approximate string searches locate strings in a database that are similar to input query strings. Forms of this technique are used in a variety of applications, such as record linkage, spell checking, and Web searches. Many use fixed  $n$ -gram-based indexing, in which fixed  $n$ -grams are substrings that have a certain length. One disadvantage of this approach is that the  $n$ -gram length can have a negative effect on the performance of the algorithms. To solve this problem, using variable length  $n$ -grams for indexing have been proposed. However, these methods have difficulty extracting variable length  $n$ -grams to use for the index. We therefore developed the algorithm, **FPI** (Frequent Pattern Indexing), to extract variable length  $n$ -grams for use with approximate string searches. **FPI** uses high-frequency patterns that appear in a dataset for index and returns exact answers. Patterns used by the proposed algorithms run in linear-time using a suffix array and an LCP array.

## General Terms

Algorithms

## 1. INTRODUCTION

Approximate string searches are used in many applications, such as spell checking, speech recognition, Web search, and record linkage.

There are many measurements between two strings, such as Jaccard similarity, cosine similarity, dice similarity, and edit distance. Above all, edit distance [9] is particularly popular due to its utility. The definition of edit distance is the minimum number of edit operations (deletion, insertion, and substitution) that are needed to transform a string to another.

In the case of the “Steve Jobs” string and the “Steven Jobs” string, if the sixth character of the latter string, ‘n’, is deleted, two strings become the same string. The edit distance is 2 in this case. However, it takes a lot of time and space to compute the edit distance, so  $n$ -gram indexing is often used to reduce the computation time. The  $n$ -grams of a string are all its substrings with a length of  $n$ . For example, the 2-gram of the string *mississippi* is {mi, is, ss, si, is, ss, si, ip, pi}. When we use this for indexing, the number of operations needed to compute the distances can be reduced because similar strings have many of the same grams. When this  $n$ -gram indexing is used, its length is often fixed. So the gram length can greatly affect its performance.

To solve this, we use the variable length  $n$ -gram for indexing. Variable length  $n$ -gram is useful for

In this paper, we propose the efficient indexing for approximate string searches using frequent patterns for Index. Using a suffix array and its LCP array, this frequent patterns are extracted in linear time. This extraction method can be used for text mining.

## 2. PROBLEM DEFINITION AND RELATED WORKS

### 2.1 Problem Definition

First, let us define some of the relevant notations used in this study. For a set  $S$ ,  $|S|$  denotes the cardinality of  $S$ . For a string  $s$ ,  $|s|$  denotes its length. For a position  $i$  ( $0 \leq i < |s|$ ),  $s[i]$  denotes the  $i$ th character in  $s$  and  $s_i$  denotes the suffix of  $s$  that starts from the  $i$ th character. For positions  $i$  and  $j$  ( $0 \leq i \leq j < |s|$ ),  $s[i : j]$  denotes the substring that starts from the  $i$ th character to the  $j$ th character. For a pair of strings  $s_1$  and  $s_2$ ,  $lcp(s_1, s_2)$  and  $ed(s_1, s_2)$  respectively denote the longest common prefix and edit distance between  $s_1$  and  $s_2$ .

In record linkage, we need to detect identical entities in a database. Entities are usually represented by a string and identical entities may be represented by slightly different strings. Approximate string matching is a fundamental function to find a candidate set of strings that may represent the identical entities. In this paper, we consider an approximate string search problem. Suppose we have a database  $D$  that

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

EDBT/ICDT '13, March 18 - 22 2013, Genoa, Italy  
Copyright 2013 ACM 978-1-4503-1599-9/13/03 ...\$15.00.

consists of numerous strings, each of which represents an entity. For a query string  $q$ , the problem is defined to obtain the following set of strings:

$$\{s \in D \mid ed(s, q) \leq k\}, \quad (1)$$

where  $k$  is a predefined threshold given by the user.

For a pair  $(s, q)$  of strings, a  $O(|s||q|)$  calculation is required and the size of the database  $D$  is usually very large. Therefore, the key point is to develop an efficient approximate string search algorithm.

## 2.2 Approximate String Matching

Approximate string matching is a longstanding problem in computer science and many types of data structures and algorithms have been proposed in response to it, such as suffix tree, automaton algorithms, and bit parallel algorithms [13][19]. However, it is still a challenging problem when handling “Big data”.

A naive solution to the approximate string search problem is to calculate the edit distance between a query string  $q$  and each string in the database. However, this is computationally prohibited when the database contains a high number of strings. We therefore usually use n-gram based indexing methods and construct an index.[18, 4, 6, 3]. An n-gram is a substring with a length of  $n$  and an n-gram-based indexing method constructs an inverted-list consisting of all available n-grams with the pointers to the records that include the n-gram. For example, let us consider a string database shown in Table 1. The bi-gram inverted index for the database is given by Table 2. Usually similar strings include common n-grams. We can solve the approximate string search problem efficiently by solving the following *T-occurrence Problem* before calculating edit distances directly[17].

Let  $G(s)$  be the multi set of the n-grams of a string  $s$ . *T-occurrence Problem* is that finding the string ids that appear at least  $T$  times on the inverted lists of grams in  $G(q)$ .

*T-occurrence Problem* can be used to filter out strings in the database whose distance is longer than the specified threshold  $k$ . It can be applied to various types of distances such as cosine similarity, Jaccard similarity, and Dice similarity. In the case of the edit distance, the number  $T$  is calculated as follows.

$$T = \max(|q|, |s|) - n \cdot (k - 1) - 1 \quad (2)$$

By using an n-gram based index, the approximate string search problem is solved by the following steps:

1. Decompose the query string into its n-grams and obtain the corresponding id lists from inverted lists.
2. Search the all strings ids that occur at least certain times ( $= T$ ) on the lists.
3. Calculate the distances or similarities between the query string and string ids obtained in step 2.

Step 2 is safe filtering to search approximate strings. Many join algorithms can be used for this, including *MergeOpt*[17], *DevideSkip*[10], and *MergeSkip*[10].

There are some disadvantages associated with n-gram based indexing. First, the performance of the search algorithms can be negatively affected by the length  $n$  grams. For a short  $n$ , string lists in inverted lists can be long, which causes longer computation time for joining the id lists. In contrast,

Table 1: string example

id	string
0	stich
1	stick
2	such
3	stuck

Table 2: inverted list example

gram	string ids
ch	→ 0,2
ck	→ 1,3
ic	→ 0,1
st	→ 0,1,2
su	→ 2
ti	→ 0,1
tu	→ 3
uc	→ 2,3

a long  $n$  generally results in huge entries in the inverted file because the number of n-grams can be  $O(|\Sigma|^n)$ , where  $\Sigma$  denotes an alphabet. Therefore, we need to choose the appropriate  $n$  carefully, or alternatively, we can use variable-length n-gram indexing method proposed in VGRAM[11].

VGRAM uses substrings with lengths between  $N_{min}$  and  $N_{max}$ , where  $N_{min}$  and  $N_{max}$  are parameters. These parameter need to be determined so that resultant grams have high frequency. However the frequency parameter are required to decide depending on the data. When we use VGRAM, we count the frequencies of the all substrings in a dataset with lengths between  $N_{min}$  and  $N_{max}$ , and if a substring is very frequent, keep some of its extended substrings for the index. This means VGRAM requires three parameters, so choosing the optimal parameters is very complicated. Moreover, it uses a tree structure for extracting the index, which causes increased time and space consumption.

When using the VGRAM index, we need to use the following  $T$  in the *T occurrence problem* [21]:

$$T = \max\{|VG(q)| - NAG(q, k), |VG(s)| - NAG(s, k)\}, \quad (3)$$

where  $VG(s)$  is the set of variable length n-grams included in  $s$ , and  $NAG(s, k)$  is the number of affected grams in  $VG(s)$  when we perform  $k$  edit operations to string  $s$ .

## 2.3 Frequent Pattern Mining

Frequent pattern mining refers to finding frequent subitemsets or sub-data structures in large databases [16]. A well-known problem is *association rule mining*. Many algorithms are in place for mining frequent sequential patterns [2], subtrees [1], or subgraphs [20], most of which are based on the *Apriori algorithm*, which prunes an itemset if its sub-set is not frequent.

Successive patterns are important, when mining frequent patterns from text data, but in the case of sub-sequential pattern mining, their algorithm can find only sequential pattern but not successive pattern. For example, *mississippi*\$ and *militarypolice*\$ have the common sequential pattern “*miipi*”. We therefore cannot apply the sub-sequential pat-

tern mining algorithms to frequent substring mining.

However some data structures have been proposed that can count the frequency of all substrings in a text [12][8]. There are also various frequent substring mining methods, including N-gram PrefixSpan [15] and Mining Frequent Substrings [22].

## 2.4 Suffix Array and LCP Array

A suffix array is a data structure that searches for any substrings that appear in the target string. Its main advantage is that it reduces the processing time and the space consumption of the suffix tree. A suffix array is constructed by sorting all suffixes in a string into a lexicographical order. Many linear-time algorithms to construct a suffix array were proposed in [7, 5, 14].

An example of the suffix array of the string “mississippi\$” is shown in TABLE 3. In a suffix array, we can efficiently count the number of times a substring  $p$  appears in the string  $T$  because all suffixes are sorted lexicographically.

Let  $i$  and  $j$  be satisfied with the following equations (in which this appearance number,  $freq_p$ , is as follows).

$$i = \operatorname{argmin}_{0 \leq a < |T|} (T_a[0; |p| - 1] = p) \quad (4)$$

$$j = \operatorname{argmax}_{0 \leq b < |T|} (T_b[0; |p| - 1] = p) \quad (5)$$

$$freq_p = j - i + 1 \quad (6)$$

In this example, when counting the substring “i” appearance,  $i = 1$  and  $j = 5$ , so  $freq_{i''} = 5 - 1 + 1 = 5$ .

An LCP array is an array that contains the length of lcp between suffixes that are adjacent in the suffix array. The LCP array of a string  $T$  is defined as follows.

$$LCP[i] = lcp(T_{SA[i]}, T_{SA[i+1]}) \quad (0 \leq i < |T| - 1) \quad (7)$$

An example of an LCP array is shown in TABLE 3. This example array is a monotonically decreasing function if one lcp pair is fixed.

$$LCP[i] \geq lcp(T_{SA_T[i]}, T_{SA_T[i+2]}) \quad (8)$$

Therefore, the lcp value of any pair in a suffix array is the least value between the interval.

$$lcp(T_i, T_j) = \min_{i \leq k < j} LCP[k] \quad (9)$$

An LCP array linear time construction algorithm has also been proposed, for use when suffix array is known[8].

## 3. PROPOSED METHOD

We propose the indexing method for approximate string matching using an LCP array: frequent pattern indexing (FPI). This method is required to extract variable length patterns for indexing, but the extraction phase requires the only linear time.

### 3.1 Frequent Pattern Indexing (FPI)

FPI is comprised of the following steps.

**Step 1** Construct the suffix and the LCP arrays of all strings in a dataset.

**Table 3: The example of suffix array and LCP array of the string “mississippi\$”**

Pos	SA	LCP	suffix
0	12	0	\$
1	11	1	i\$
2	8	1	ippi\$
3	5	4	issippi\$
4	2	0	ississippi\$
5	1	0	mississippi\$
6	10	1	pi\$
7	9	0	ppi\$
8	7	2	sippi\$
9	4	1	sissippi\$
10	6	3	ssippi\$
11	3	-	ssissippi\$

**Step 2** Run algorithm 1 in Fig. 1 and extract frequent patterns in a dataset.

**Step 3** Index a dataset using frequency patterns.

In step 1, we concatenate all strings in a dataset,  $D$ , with a special end mark, ( $\$ \notin \Sigma$ ), make one string, and compute the suffix array and LCP array of this string.

In step 2, Algorithm 1 computes the frequent patterns whose frequency is equal or longer than the specified parameter  $n$ . To compress the index, this algorithm output only the maximal substrings whose frequency is equal or longer than  $n$ . Suppose a frequency of a substring “abcde” is equal or longer than  $n$ . Then, it is obvious that frequencies of its substrings such as “abc” and “abcd” is equal or longer than  $n$ . If frequencies of any strings that include “abcde” are less than  $n$ , i.e., “abcde” is optimal, the algorithm outputs only “abcde” and omits its substrings. This algorithm requires a LCP array, its size( $N$ ) and just two parameters for the input: the minimum frequency pattern length ( $n$ ) and the frequency parameter( $\tau$ ).

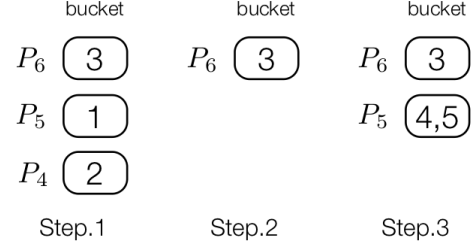
The variables and the functions used in Algorithm 1 are denoted as follows.

- *candidate* : the output candidate that contains a position and an LCP value,
- $P_k$  : the bucket of positions with an LCP value of  $k$ ,
- $len(P_k)$  : return of the number of positions in  $P_k$ ,
- $P.clean$  : removal of all buckets,
- *candidate.initialled* : the setting of both candidate. position and candidate. lcp to 0

In the algorithm, first, set  $tmp_n$  to  $n$ . Second, the positions at which LCP values of more than  $tmp_n$  are the same should be placed in the same bucket until the number of positions in the all buckets reaches  $\tau$ . If a value less than  $tmp_n$  is found, remove the all buckets and, if a candidate remains, output the candidate. This removal is can be performed safely because of Eq. (9). When the volume of the bucket contents reaches  $\tau$ , select the minimum value from the LCP values of the buckets, and update the candidate, buckets and  $tmp_n$ . Because patterns that function as the prefixes of other frequency patterns must not be outputted,

**Table 4: The example of LCP array**

Pos	0	1	2	3	4	5	6	7	8	9	...
LCP	0	5	4	6	5	5	6	3	4	2	...



**Figure 1: The example of Applying Algorithm 1**

---

**Algorithm 1** Extracting frequent substrings

---

```

1: Input: LCP, N, n and  $\tau$ 
2:  $i := 1$ 
3:  $e := \tau$ 
4:  $tmp_n := n$ 
5: candidate.initialled
6: while  $e < N$  do
7:   while  $(\sum_k len(P_k) < \tau \text{ and } e < N)$  do
8:     if  $LCP[i] < tmp_n$  then
9:       P.clean
10:       $tmp_n := LCP[i]$ 
11:       $i++$ 
12:       $e := i + \tau$ 
13:      if  $candidate.lcp > n$  then
14:        Output:candidate
15:        candidate.initialled
16:      end if
17:    else
18:       $P_{LCP[i]}.push(i)$ 
19:       $i++$ 
20:    end if
21:  end while
22:   $m \leftarrow$  the minimum value  $k$  of  $P_k$ 
23:   $s \leftarrow$  the last value in  $P_m$ 
24:  for all  $P_k$  do
25:     $P_k.pop()$  until the first value in  $P_k$  becomes the least
    number which is greater than  $s$ 
26:  end for
27:  if  $candidate.lcp < m$  then
28:     $candidate.lcp := m$ 
29:     $candidate.position := s$ 
30:     $tmp_n := m$ 
31:  end if
32: end while
33: if  $candidate.lcp > n$  then
34:   Output:candidate
35: end if

```

---

the output function is implemented only when values less than that of the candidate are found.

An example of behavior of this algorithm using the LCP array from Table 4 and  $\tau = 3, n = 3$  is shown in Fig. 1. When the position goes to 3, the buckets 4, 5, and 6 are created and put in their corresponding positions (step 1 in Fig.1). In this case, since the number of positions reaches the  $\tau$ , the least number of the buckets, 4, is set to the candidate (*candidate.position*= 2, *candidate.value*= 4) and all positions in the all buckets less than *candidate.position* (step 2 in Fig.1) are removed. When the position reaches 5, the candidate is updated to *candidate.position*= 5, *candidate.value*= 5 because of the previous candidate.value is less than the present value. Since the LCP value is less than the candidate. value, the candidate is output when the position moves to 7.

In step 3 of FPI, index the all strings in a dataset are indexed using the frequency patterns obtained in step 2. *FP* denotes these pattern sets. To reduce the size of the inverted index, impose the following rules.

1. Use the longest substrings with which any prefix of patterns in *FP* coincide.
2. Do not use any substrings that belongs to another index.

For example, if  $FP = \{\text{'uni'}, \text{'iv'}, \text{'ivr'}, \text{'ver'}, \text{'vers'}, \text{'rs'}, \text{'sit'}, \text{'ty'}, \text{'ity'}\}$  the index set of the string "university" is not  $\{\text{'uni'}, \text{'iv'}, \text{'ver'}, \text{'rs'}, \text{'sit'}, \text{'ity'}, \text{'ty'}\}$  but rather  $\{\text{'uni'}, \text{'iv'}, \text{'vers'}, \text{'sit'}, \text{'ity'}\}$ . In spite of the existence of the 'vers' pattern, the set does not use the longest substring for the index, and 'ty' is the substring of 'ity' so the index set becomes the latter. If no substring is found in *FP*, we use the length  $n$  substring for the index, instead.

We process the query the same as in the  $n$ -gram-based indexing. However, when decomposing the query, we use *FP*.

## 4. EXPERIMENTAL RESULTS

In this section, we present the results of our experiments with the *FPI* and the *LFPI* methods. First, we measured

<sup>1</sup><http://www.sisap.org>

<sup>2</sup><http://www.informatik.uni-trier.de/~ley/db/index.html>

**Table 5: Data sets used in the Experiments**

Data	Size	Averaged length	$\Sigma$	Description
EnglishDictionary	635K	9.4	26	english dictionary supplied in SISAP <sup>1</sup>
DBLP	78M	104.5	93	bibliography data supplied in DBLP <sup>2</sup>

**Table 6: Data sets used in the Experiments**

	Data	parameters	Extraction(sec)	Query Processing(sec)
FPI	Englism	$(n, \tau) = (2, 200)$	0.02	2.59
VGRAM	English	$(N_{min}, N_{max}, \tau) = (2, 4, 1000)$	0.17	2.40
FPI	DBLP	$(n, \tau) = (4, 1000)$	10.6	173.0
VGRAM	DBLP	$(N_{min}, N_{max}, \tau) = (4, 8, 10000)$	194.4	176.8

**Table 7: Suffix array and LCP array construction Time**

Dataset	Suffix Array(sec)	LCP array(sec)
English Dictionary	0.3	0.09
DBLP	96.81	25.36

the patterns extraction time of both algorithms. For this extraction, we used the `sa_is` algorithm[14] to construct the suffix array, and used the linear-time LCP computation algorithm[8] to construct the LCP array. Second, we evaluated the effect of the different parameters on the performance of our algorithms. In query processing in FPI, we used the MergeOpt algorithm[17] to merge the string ids lists.

We performed these experiments on a server running Red Hat GNU/Linux 5.1 with an AMD Opteron(TM) 8384 CPU (2.70GHz) and 192 GB main memory. All of algorithms are written in C and we compiled all codes with a GCC compiler.

## 4.1 Datasets

The three datasets we used are shown in TABLE5. DBLP records contain the journal title, author, conference names, etc. We concatenated all the values in one record and then constructed one string from this record.

For query processing, we generated the random queries: for each dataset, we randomly chose 1,000 strings and then randomly made some minor changes to each string. Using the “English Dictionary” dataset, we randomly performed one edit operation on selected strings and made queries, while for the “DBLP”, we performed five edit operations and made queries. In query processing, we measured how long it took to retrieve all the strings corresponding to the queries from the datasets.

We compared with VGRAM. In suffix array indexing, first, we construct a suffix array of the string that we concatenate all strings in a dataset. In query processing, we search the longest common prefix between a query strings and strings using suffix array and calculate the edit distance between a query and strings that have the prefix. This suffix array indexing doesn’t also find exact answers.

## 4.2 Results

**Index Extraction :** We evaluated the overhead of the proposed methods. We fixed the pattern length to  $n$  and

varied the frequency parameter  $\tau$  for each datum. In English Dictionary, we chose  $n = 2$  and varied  $\tau$  from 100 to 250. In DBLP,  $n = 4$  and  $\tau$  was varied from 1000 to 5000. TABLE7 shows the time it took to construct both suffix and LCP arrays for each datasets. The pattern extraction times are shown in Fig.2(FPI) and in Fig. The larger the frequency parameter is, the more time it takes.

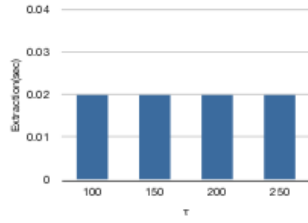
**The Effect of the Parameter  $n$  in FPI:** We examined the effect of the parameter as pattern length in FPI method, using the English Dictionary and DBLP datasets. With English Dictionary,  $\tau = 200$  and  $n$  was changed from 2 to 5 and with DBLP,  $\tau = 1500$  and  $n$  was changed from 3 to 8. The results are show in Fig.3. When we used English dictionary, the query processing time increased as  $n$  increased, while DBLP, as  $n$  increased, the query processing time first decreased, then reached a minimum at  $n = 6$  (DBLP), and then increased again. This is because when  $n$  is large, the number of the substrings shared by various strings can be small and the string lists thus easily merged, but when the  $n$  is too large,  $T$  can be less than 0 and the system needs to check all strings in a dataset.

**The Effect of the Parameter  $\tau$  in FPI:** We next examined the effect of the frequency parameter  $\tau$  using the same datasets. We set  $n = 2$  and varied  $\tau$  from 100 to 250 for English Dictionary, and  $n = 8$  and varied  $\tau$  from 1000 to 5000 for DBLP. The results are shown in Fig.4. The best performances were shown at  $\tau = 200$  (English Dictionary) and  $\tau = 1500$  (DBLP). This is because the averaged index length becomes long in small  $\tau$  and  $T$  in *T-occurrence Problem* can be less than 0. However, in large  $\tau$ , almost all index lengths can be  $n$ , and this approximate string search system can’t benefit from variable length N-grams.

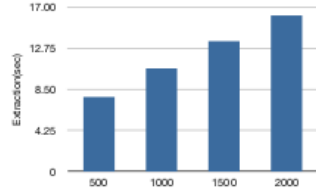
**Comparison FPI vs. VGRAM** We compared FPI with VGRAM. We examined both of variable length n-gram extraction time and query processing time. The results and parameters that we used are shown in TABLE6. The variable length extraction time in FPI was about 10 times faster than VGRAM and the query processing time was nearly equal.

## 5. CONCLUSION

In this paper, we presented new indexing structures **FPI** for approximate string matching. The pattern extraction algorithms used in **FPI** can also be used for text mining. These algorithms can run in a linear-time using a suffix array and an LCP array.

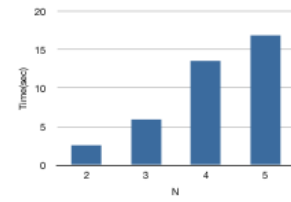


(a) English Dictionary

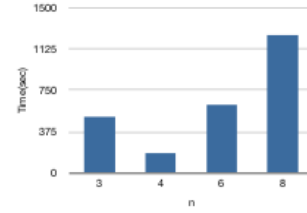


(b) DBLP

Figure 2: Index Extraction Time(FPI)



(a) English Dictionary

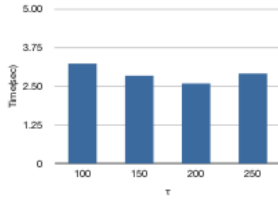


(b) DBLP

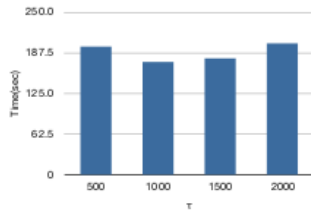
Figure 3: Query Performance(n)

## 6. REFERENCES

- [1] Kenji Abe, Shinji Kawasoe, Tatsuya Asai, Hiroki Arimura, and Setsuo Arikawa. Optimized substructure discovery for semi-structured data. In Tapio Elomaa, Heikki Mannila, and Hannu Toivonen, editors, *Principles of Data Mining and Knowledge Discovery*, volume 2431 of *Lecture Notes in Computer Science*, pages 57–100. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-45681-3.
- [2] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. pages 3–14, 1995.
- [3] Alexander Behm, Shengyue Ji, Chen Li, and Jiaheng Lu. Space-constrained gram-based indexing for efficient approximate string search. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 604–615, Washington, DC, USA, 2009. IEEE Computer Society.
- [4] Marios Hadjieleftheriou and Chen Li. Efficient approximate search on string collections. *Proc. VLDB Endow.*, 2(2):1660–1661, 2009.
- [5] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, November 2006.
- [6] Min-Soo Kim, Kyu young Whang, Jae-Gil Lee, and Min jae Lee. n-gram/2l: A space and time efficient two-level n-gram inverted index structure. In *In VLDB*, pages 325–336, 2005.
- [7] Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2&A\$4):143 – 156, 2005. Combinatorial Pattern Matching (CPM) Special IssueThe 14th annual Symposium on combinatorial Pattern Matching.
- [8] Gad Landau, Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Combinatorial Pattern Matching*, volume 2089 of *Lecture Notes in Computer Science*, pages 181–192. Springer Berlin / Heidelberg, 2006. 10.1007/3-540-48194.
- [9] Vladimir I Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission*, 1(1):8–17, 1965.
- [10] Chen Li, Jiaheng Lu, and Yiming Lu. Efficient merging and filtering algorithms for approximate string searches. *Data Engineering, International Conference on*, 0:257–266, 2008.
- [11] Chen Li, Bin Wang, and Xiaochun Yang. Vgram: improving performance of approximate queries on string collections using variable-length grams. In *VLDB ’07: Proceedings of the 33rd international conference on Very large data bases*, pages 303–314. VLDB Endowment, 2007.
- [12] Makoto Nagao and Shinsuke Mori. A new method of n-gram statistics for large number of n and automatic extraction of words and phrases from large text data of japanese. In *Proceedings of the 15th conference on Computational linguistics - Volume 1, COLING ’94*, pages 611–615, Stroudsburg, PA, USA, 1994. Association for Computational Linguistics.
- [13] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33:31–88, March 2001.
- [14] Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Transactions on Computers*, 60:1471–1484, 2011.
- [15] Jian Pei, Jiawei Han, B. Mortazavi-Asl, H. Pinto,



(a) English Dictionary



(b) DBLP

**Figure 4: Query Performance( $\tau$ )**

Qiming Chen, U. Dayal, and Mei-Chun Hsu. Prefixspan,: mining sequential patterns efficiently by prefix-projected pattern growth. In *Data Engineering, 2001. Proceedings. 17th International Conference on*, pages 215 –224, 2001.

- [16] Anand Rajaraman and Jeffrey D Ullman. Mining of massive datasets. *Lecture Notes for Stanford CS345A Web Mining*, 67(3):328, 2011.
- [17] Sunita Sarawagi and Alok Kirpal. Efficient set joins on similarity predicates. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 743–754, New York, NY, USA, 2004. ACM.
- [18] Esko Ukkonen. Approximate string matching with q-grams and maximal matches. Technical report, 1991.
- [19] Chuan Xiao, Wei Wang, and Xuemin Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *Proc. VLDB Endow.*, 1(1):933–944, 2008.
- [20] Xifeng Yan and Jiawei Han. gspan: graph-based substructure pattern mining. In *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*, pages 721 – 724, 2002.
- [21] Xiaochun Yang, Bin Wang, and Chen Li. Cost-based variable-length-gram selection for string collections to support approximate queries efficiently. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 353–364, New York, NY, USA, 2008. ACM.
- [22] TSUBOI Yuta. Mining frequent substrings(natural language understanding and models of communication). *IEICE technical report. Natural language understanding and models of communication*, 103(408):79–86, 2003-10-31.