AN IMPROVED DYNAMIC DICTIONARY MATCHING USING INVERTED LISTS

Chouvalit Khancome and Veera Boonjing
Software Systems Engineering Laboratory
Department of Mathematics and Computer Science
Faculty of Science
King Monkut's Institute of Technology at Ladkrabang(KMITL)
Ladkrabang,Bankok 10520,THAILAND
E-mail: chouvalit@hotmail.com , kbveera@kmitl.ac.th

ABSTRACT

This paper proposes to improve a dynamic dictionary matching using inverted lists which employs inverted lists as data structures accommodating string patterns. The new solution takes (1) O(|P|) times for preprocessing, where |P| is a sum of the length of all patterns in set of pattern P; (2) O(|p|) times for insertion or deletion, where |p| is the length of pattern to be inserted or deleted, (3) O(m+σ) times in average search, and (4) O(|t|) times in worst case, where m is the length of the longest pattern in P, σ is the number of occurrences of matching that lead to mismatched and included the mismatched times, and |t| is the length of input text.

KEY WORDS
Dictionary matching, inverted index, inverted list, trie, pattern

1. Introduction

The problem of dynamic dictionary matching is to efficiently locate a set of patterns occurring in an input text. In this problem, the set of patterns can change over time because of insertion and deletion of individual patterns. It calls for a data structure accommodating this set, which (1) allows quick insertions of patterns into the dictionary as well as deletions of patterns from the dictionary and (2) supports efficient searching for pattern strings in the input text. A trie, used by fast dictionary matching solutions such as [1], [9], [12], is an example of data structure supporting such an efficient searching. Unfortunately, insertions and deletions of patterns require reconstruction of the trie [2], [3], [4], [5], [6], [7], [8]. Solutions to these problems are the modifications of trie such as a suffix tree [2], [3], [11], 14] and a combination of a compact Trie and a fat tree [16].

The new ideas [17], [18], [19], [24] used inverted lists as new data structure which derived from an inverted index used in information retrieval field [10], [13], [15], instead of using a trie or a trie-based data structure. Furthermore, it well supports dynamic patterns. Especially, the inverted list in [17], [24] are very simple and highly efficient in preprocessing phase, insertion and deletion patterns. However, the searching phase has more time complexity. In this paper, we propose to improve searching and modify other parts of that algorithm.

The rest of paper is organized as follows. Section 2 gives preliminaries of inverted lists dictionary. Section 3 describes the inverted list dictionary its update algorithms as well as their proofs of time complexity. Section 4 shows a new efficient search algorithm and conclusion is in section 5.

2. Preliminaries

This section has shown the basic definition, the examples and the efficient in [17], [24].

Let $P=\{p^1,p^2,\ldots,p^r\}$ where p^i is a string from $c_1c_2,\ldots,c_m$ under $\sum$ and $\sum$ is the set of the character in P.

2.1 Basic Definitions

Definition 1 A keyword ω^i of pattern p^i contains $w_{a_{1,0,i}}\ w_{b_{2,0,i}}\ w_{c_{3,0,i}}\ \ldots\ w_{\cdots m,1,i}$; where $w_{n_{k,0,i}}$ or $w_{n_{k,1,i}}$ is c_k and k = 1, 2, …, m; 1 indicates a status of last character in p^i and 0 otherwise. Therefore,

$$\omega^i = w_{a_{1,0,i}}\ w_{b_{2,0,i}}\ w_{c_{3,0,i}}\ \ldots\ w_{\cdots m,1,i} \quad (1)$$

Example 1 Suppose P={aab, aabc, aade}. We have ω^1=aab, ω^2=aabc and ω^3=aade. Therefore,
$\omega^1=a_{1,0,1}a_{2,0,1}b_{3,1,1,}$

ω^2=$a_{1,0,2}a_{2,0,2}b_{3,0,2}c_{4,1,2,}$ and
ω^3=$a_{1,0,3}a_{2,0,3}d_{3,0,3}e_{4,1,3,}$

Definition 2 An inverted list L of ω^i , denoted by L_{ω^i}, is defined as

$$L_{\omega^i} = w_a :< 1:0:i >, w_b :< 2:0:i >, w_c :< 3:0:i >,..., w_{...} :< m:1:i > \quad (2)$$

Example 2 From example 1, we have

L_{ω^1} = a:<1:0:1>, a:<2:0:1>, b:<3:1:1>,

L_{ω^2} = a:<1:0:2>, a:<2:0:2>, b:<3:0:2>, c:<4:1:2>, and

L_{ω^3} = a:<1:0:3>, a:<2:0:3>, d:<3:0:3>, e:<4:1:3>.

Definition 3 An index w_λ of invert list is $< \varepsilon : 0 : \{i, j,...\} >$ or $< \varepsilon : 1 : \{i\} >$. Therefore,

$$w_\lambda : < \varepsilon : 0 : \{i, j,...\} > \text{ or } < \varepsilon : 1 : \{i\} > \quad (3)$$

Example 3 From example 2, we have

a : <1:0:{1,2,3}>, <2:0:{1,2,3}>,
b: <3:1:{1}>, <3:0:{2}>,
c: <4:1:{2}>,
d: <3:0:{3}>, and
e: <4:1:{3}>.

Definition 4 Let $I_{\lambda_{\varepsilon,0}}$ and $I_{\lambda_{\varepsilon,1}}$ be $< \varepsilon : 0 : \{i, j,...\} >$ and $< \varepsilon : 1 : \{i\} >$, respectively. Therefore,

$$w_\lambda : I_{\lambda_{\varepsilon,0}} \text{ or } w_\lambda : I_{\lambda_{\varepsilon,1}}. \quad (4)$$

Definition 5 An inverted list table τ is a hash table with 2 columns: w_λ and $I_{\lambda_{\varepsilon,0}} / I_{\lambda_{\varepsilon,1}}$; where w_λ contains w_a, and $I_{\lambda_{\varepsilon,0}} / I_{\lambda_{\varepsilon,1}}$ contains $I_{a_{1,0}}, I_{a_{2,0}}, I_{a_{5,1}}, \ldots$. (5)

Example 4 The table τ constructed from example 3 is as shown in table1.

Table 1 the table of pattern P={aab,aabc,aade}

w_λ	$I_{\lambda_{\varepsilon,0}} / I_{\lambda_{\varepsilon,1}}$
a	<1:0:{1,2,3}><2:0:{1,2,3}>
b	<3:1:{1}><3:0:{2}>
c	<4:1:{2}>
d	<3:0:{3}>
e	<4:1:{3}>

Theorem 1 The access to $I_{\lambda_{\varepsilon,0}}$ or $I_{\lambda_{\varepsilon,1}}$ in the table τ take O(1) time.

Proof Let f(x) be a hash function let $w_{\lambda_{\varepsilon,0}}$ be the key for access $I_{\lambda_{\varepsilon,0}}$ and $w_{\lambda_{\varepsilon,1}}$ be the key for access $I_{\lambda_{\varepsilon,1}}$.

Let the table τ implemented by the hash table as [20], [21], [22], [23] that takes O(1), therefore the access to $I_{\lambda_{\varepsilon,0}}$ with f($w_{\lambda_{\varepsilon,0}}$) or access $I_{\lambda_{\varepsilon,1}}$ with f($w_{\lambda_{\varepsilon,1}}$) takes O(1) time ■

The solution of [17] and [24] take (1) O(|P|) times for preprocessing, where |P| is a sum of the length of all patterns in set of pattern P; (2) O(|p|) times for insertion or deletion, where |p| is the length of pattern to be inserted or deleted; and (3) a search O(|t|+locc) times, where |t| is the length of input text and locc is the number of occurrences of matching between a character in the input text and in the inverted list. However, this solution takes O(|t||P|) in worst cases of searching phase. So, this paper proposes to improve this searching as follows.

3. Inverted List Dictionary

This section explained new algorithms which consisted preprocessing phase for creating dictionary, insertion and deletion patterns from dictionary.

3.1 Preprocessing phase

Let $\sum$ be a set of all characters in P and char(p_j^i) be a character 'j' of the pattern 'i'. Let |P| be a total length of all patterns in P. Before adding all character from $\sum$ to a character column, we must create the inverted list table and dedicate min_length to maximum of integer. Afterwards, this algorithm reads a character one by one from each pattern and adds into the inverted list column. Before the addition, there is a process to check the inverted list char(p_j^i) at the same position. If it already exists the inverted list of the char(p_j^i) at the same position then we add only the number pattern to the invert list table. Otherwise, we must create a new inverted list and add it into the table. After all patterns are inserted, we must set m to min_length if m less than current min_length. Figure 1 gives details of the algorithm.

Inverted-List Table(P={p^1,p^2,…,p^r})
Step A Create table for alpha from $\sum$ and min_length = maximum of integer number
Step B for $i:1:r$
Step C for $j:1:m$
Step D if not exist inverted list of p_j^i
Step D1 Generate a new invert list and add to table at alphabet char(p_j^i)
else
Step D2 Add a number *i* to the set of the represent part of the invert list at alphabet p_j^i
Step E set m to min_length if min_length > m

Fig. 1 Algorithm for creating inverted list table

Theorem 2 The time complexity of preprocessing phase algorithm is O(|P|) .
Proof Given P={p^1,p^2,p^3,…p^r} and the length of each of p^i is m_i such $m_1+m_2+m_3+\ldots+m_r = |P|$.
Step A create table for storing the inverted list takes O(1) time.
Step B repeat each of p^i takes r rounded. Each round must loop in Step C m_j times. Therefore, the inverted list takes $m_1+m_2+m_3+\ldots+m_r = O(|P|)$ times.
Step D, Step D1 and Step D2 use for checking in table that takes O(1) by theorem 1.
Step E take O(r), but r is the total pattern and it less than O(|P|).
Therefore, the time complexity of this algorithm is O(|P|). ■

3.2 Pattern insertion

Let p^i be a new pattern for insertion where i is a number refer to unique symbol. The insertion must check the non-existence of p^i in the table. Afterwards, reading and inserting a character from pattern into table which is similar to the preprocessing phase. Importantly, we must set minimum of pattern length as preprocessing phase. This algorithm is illustrated by the figure 2.

InsertPattern(p^i)
Step A if not Exist(p^i)
Step B for j:1:m
if not exist of char(p_j^i)
Step B1 Generate a new invert list and add to table at alphabet char(p_j^i)
else
Step B2 Add a number *i* to the set of the represent part of inverted list at alphabet p_j^i
Step C set m to min_length if min_length>m

Fig. 2 Algorithm for pattern insertion

Theorem 3 Time complexity of the algorithm for pattern insertion is O(|p|).

Proof Let p^i be a new pattern for insertion where p^i contains a string $p^i = c_1c_2c_3\ldots c_m$ such that the length m represented by |p|.

Step A Repeating from c_1 to c_m, use m time that is |p| or take O(|p|) times. The access to the inverted list table takes O(1) followed by theorem 1.

Step B Repeating for adding pattern one by one from c_1 to c_m that use |p|, therefore that takes O(|p|) times.

Step B1 or B2 access to table taking constant time by theorem 1. That all of Step B takes O(|p|) times.

Step C takes O(1) because it takes only one time in each of pattern p.

Therefore, the insertion algorithm takes O(|p|) times. ■

3.3 Pattern deletion

Let p be a pattern for deletion and 'Numberpattern' be a number of pattern. ExistDel(p) is a function to detect the existence of pattern for deletion. The deletion must search for pattern p using ExistDel(p) function and the result is the pattern number for deletion. It then searches one by one for deletion until finish. The importance of deletion is that we need to check the inverted list in the same position of a number pattern that we want to delete. If it has only one, we can delete that inverted list immediately. Otherwise, we must delete an inverted list only 'Numberpattern'. After deleted each of patterns, we set min_length as preprocessing. The algorithm is illustrated by figure 3.

DeletePattern(p)
Step A Numberpattern = ExistDel(p)
if Numberpattern != θ
Step B for i:1:m
Search inverted list of $p_i^{Numberpattern}$
if number of the items in represent pattern $p_i^{Numberpattern} > 1$
Step B1 Delete items in represent part = numberpattern
else
Step B2 Delete inverted list of $p_i^{Numberpattern}$
Step C set min_length to the less than prior if m is the minimum pattern length

Fig. 3 Algorithm for pattern deletion

Theorem 4 The deletion algorithm takes O(|p|) times.

Proof Let p^i be a pattern for deletion where p^i contains a string $p^i = c_1c_2c_3\ldots c_m$ with length m = |p|.

Step A repeat to read from c_1 to c_m takes O(|p|) times. Each time we access an inverted list use O(1) by theorem 1. Therefore, this step takes O(|p|) times.

Step B repeat for read a character one by one from c_1 to c_m takes O(|p|) times.

Step B1 or B2 accesses the inverted list with constant time from theorem 1. It takes O(|p|) time.

Step C take only once, hence, it take O(1) after deleted the request pattern.

Therefore, the deletion algorithm takes O(|p|) time. ■

4. Searching phase

The searching phase employs the navigator variable N as the current comparison position; SHIFT as the shift window; and SET1, SET2, and SETE as the temporary variables used in matching.

The first character of each search window is compared with the last character in the text followed by taking the inverted list to SETE for reference. If SETE is not empty and matches with the last character, we scan to compare the text from the first to the last character, or if SETE does not contain the last character, we consider the farthest character matching the SETE and scan from suitable position which matched character in that windows. Every comparison takes the inverted list to the temporary variable SET1 or SET2, meanwhile taking the inverted list to these variables. We must also operate SET1 and SET2. The purpose of the operation is to search for the sequence of pattern and check the matching. We illustrate the algorithm in figure 4.

Lemma 1 Let SET be the sub table with keys $w_{\lambda_{\varepsilon,0}}$ and $w_{\lambda_{\varepsilon,1}}$ for accessing $I_{\lambda_{\varepsilon,0}}$ and $I_{\lambda_{\varepsilon,1}}$, respectively. The access to $I_{\lambda_{\varepsilon,0}}$ and $I_{\lambda_{\varepsilon,1}}$ in SET using f($w_{\lambda_{\varepsilon,0}}$) or f($w_{\lambda_{\varepsilon,1}}$) function takes O(1) times.

Proof Let SET be the hashing a table with keys $w_{\lambda_{\varepsilon,0}}$ and $w_{\lambda_{\varepsilon,1}}$.

Therefore, accessing to $I_{\lambda_{\varepsilon,0}}$ and $I_{\lambda_{\varepsilon,1}}$ using f($w_{\lambda_{\varepsilon,0}}$) and f($w_{\lambda_{\varepsilon,1}}$) takes O(1) times by theorem 1. ■

Inverted-List Mutiple-Pattern Search ($P=\{p^1,p^2,p^3,\ldots,p^r\}, T=t_1t_2\ldots t_n$)
Preprocessing Phase :
Create Inverted-List Table($P=\{p^1,p^2,p^3,\ldots,p^r\}$)
Searching Phase
Step A N=min_length ,SHIFT=2x(min_length), SET1={},SET2={},SETE={}
Step B While N<n and SHIFT<=n
Step C SETE←inverted lists of text[N] and N=the farthest position from set SETE
Step D SET1← inverted lists of text[N]
Step E While SET1 !=Empty
Step E1 SET2← inverted lists of text[N] if text[N] is not position of SETE
Step E2 SET1 ← SET1 operate SET2/or SET1 ← SET1 operate SETE if position of text[N] is SETE position meanwhile operation must be check matching if terminate status = 1 and remove that list from SET1 and Update SHIFT if its position beyond current window search
Step E3 If SET1 != Empty set value N← N+1
Step F N=SHIFT, SHIFT←SHIFT+min_length,

Fig. 4 Algorithm for searching

Lemma 2 To take the inverted list from τ matching text[N] into SET takes O(1) times.

Proof Let text[N] be a character from string T with keys $w_{\lambda_{pos,0}}$ and $w_{\lambda_{pos,1}}$. The access to $I_{\lambda_{pos,0}}$ and $I_{\lambda_{pos,1}}$ in table τ takes O(1) times.

Therefore, to take $I_{\lambda_{pos,0}}$ and $I_{\lambda_{pos,1}}$ into SET takes O(1) times by lemma 1. ■

Definition 6 An operation for continuity from position $\varepsilon 1$ to $\varepsilon 2$ of $I_{\lambda q_{\varepsilon 1,0}}$ and/or $I_{\lambda q_{\varepsilon 1,1}}$ in SET1 and $I_{\lambda b_{\varepsilon 2,0}}$ and/or $I_{\lambda b_{\varepsilon 2,1}}$ in SET2 is a set of pattern numbers that the character described by SET2 follows the character described by SET1. (6)

Definition 7 An inverted list $<1:0:\{i,j,...\}>$ or $<1:1:\{i,j,...\}>$ of SET2 always continues from SET1. (7)

Example 5 Suppose SET1 ={<**1**:0:{**1**,2}>} and SET2={<**2**:0:{**1**,3}>,<1:0:{5}>}. The operation for continuity from position 1 to 2 of SET1 and SET1 is {**1**}. Therefore, the character described by SET2 follows the character described by SET1 in pattern number 1 and moreover, the inverted list <1:0:{5}> also continues from SET1 by definition 7.

Lemma 3 The operation between SET1 and SET2 takes O(1) times.

Proof Let SET1 be a set of $I_{\lambda_{\varepsilon 1,0}}$ and/or $I_{\lambda_{\varepsilon 1,1}}$, and SET2 be a set of $I_{\lambda_{\varepsilon 2,0}}$ and /or $I_{\lambda_{\varepsilon 2,1}}$.

The access to $I_{\lambda_{\varepsilon 1,0}}$, $I_{\lambda_{\varepsilon 1,1}}$, $I_{\lambda_{\varepsilon 2,0}}$ and $I_{\lambda_{\varepsilon 2,1}}$ for operation takes O(1) times by lemma 1.■

Theorem 5 The search algorithm takes O(m+σ) times.

Proof

Average case: Let |t| be the sum of length of $T=t_1t_2t_3\ldots t_n$, σ be the number of occurrences of matching that leads to mismatched and included the mismatched times, and m is the longest pattern in $P=\{p^1,p^2,p^3,\ldots,p^r\}$.

Step A initializes variables take O(1) times. The inner of time complexity happens from step E which is the each of windows search and steps within step E covers step E1 to step E3 takes O(m) time in the matched case or take O(σ) in the mismatched case, meanwhile in each operation of them to take text[N] uses O(1) by lemma 2. Whole operations of SET according to definition 6 and definition 7 take O(1) by lemma 1. Step B is the external loop. Step C, D and F depend on step E in the case of 1) matched all windows search take O(m)+O(m)+O(m) +….O(m), 2) mismatched all windows search take O(σ)+O(σ)+O(σ)+O(σ)+…, and 3) mismatched in some windows search take O(m)+O(σ)+O(σ)+ O(m)+…. Therefore, the maximum of time complexity of searching phase is O(m+σ).■

Worst case: The happening of worst case of this algorithm, if the first window scan the text in the case of SET1 could not empty if only once. It works in step C one by one character from the beginning to the end of text and it takes completely O(|t|). ■

5. Conclusion

This paper presents an improved solution to the dynamic dictionary matching using inverted lists. We show that this solution takes (1) O(|P|) times for preprocessing, where |P| is a sum of the length of all patterns in set of pattern P; (2) O(|p|) times for insertion or deletion, where |p| is the length of pattern to be inserted or deleted; and (3) average search takes O(m+σ) times, and (4) worst case search takes O(|t|) times, where m is the length of the longest pattern in P, σ is the number of occurrences of matching that lead to mismatched and included the mismatched times, and |t| is the length of input text.

References

[1] A. V. Aho and M. J. Corasick. "Efficient string matching. An aid to bibliographic search". *Comm. ACM*, 1975. 333-340.

[2] A. Amir and M. Farach. "Adaptive dictionary matching". *Proc. of the 32nd IEEE Annual Symp. On Foundation of Computer Science*, 1991. 760-766.

[3] A. Amir, M. Farach, R.M. Idury, J.A. La Poutr'e, and A.A. Schaffer. "Improved Dynamic Dictionary-Matching". *In Proc. 4nd ACM-SIAM Symp. on Discrete Algorithms*. 1993, 392-401.

[4] A. Amir, M. Farach, R. M. Idury, J. A. La Poutré , and A. A. Schäffex. "Improve dynamic dictionary matching". *Information and Computation, 199(2)*. 1995, 258-282.

[5] A, Amir, M. Farach, and Y. Matias. "Efficient randomized dictionary matching algorithms". *In CPM: 3rd Symposium on Combinatorial Pattern Matching*, 1992.

[6] A. Amir, M. Farach, Z. Galil, R. Giancarlo, and K. Park. "Dynamic dictionary matching". *Manuscript*. 1991.

[7] A. Amir, M. Farach, Z. Galil, R. Giancarlo, and K. Park. "Dynamic dictionary matching". *Journal of Computer and System Sciences*. 1993.

[8] A. Amir, M. Farach, Z. Galil, R. Giancarlo, and K. Park. "Dynamic dictionary matching". *Journal of Computer and System Science, 49(2)*. 1994, 208-222.

[9] B. Commentz-Walter. "A string matching algorithm fast on the average". *In Proceedings of the Sixth International Collogium on Automata Languagees and Programming*. 1979, 118-132.

[10] C. Monz and M. de Rijke. (2002) *Inverted Index Construction*. [Online]. Available : http://staff.science.uva.nl/~christof/courses/ir/transparencies/clean-w-05.pdf .

[11] D. D. Sleator and R. E. Tarjan. "A data structure for dynamic trees". *Journal of Computer and System Sciences 26(3)*. 1983, 362-391.

[12] G. Navarro and M. Raffinot. "*Flexible Pattern Matching in Strings*". The press Syndicate of The University of Cambridge. 2002.

[13] O. R. Zaïane. "CMPUT 391: Inverted Index for Information Retrieval". *University of Alberta*. 2001.

[14] P. F. Dietz and D. D. Sleator. "Two algorithm for maintaining order in a list". *In Proceeding of the Nineteenth Annual ACM Symposium on Theory of Computing*. 1997, 365-372.

[15] R. B. Yates and B. R. Neto. "Mordern Information Retrieval". *The ACM press.A Division of the Association for Computing Machinery,Inc*. 1999, 191-227.

[16] S. Sahinalp and U. Vishkin. "Efficient approximate and dynamic matching of patterns using a labeling paradigm (extended abstract)". *In 37th Annual Symposium on Foundations of Computer Science*. 1996, 320-328.

[17] C. Khancome and V. Boonjing. Inverted-List Multiple String-Pattern Matching. *Aplied Statistics Conference 2006*, NIDA, 2006, 287-296.

[18] C. Khancome and V. Boonjing. String Matching Using Inverted List. *XIX International Conference on Computer, Information and Systems Science and Engineering (CISE 2007)*. Enformatika Transcations on Engineering, Computer and Technology Quarterly volumn 19, January, 2007, 108-112.

[19] C. Khancome and V. Boonjing. String Matching Using Inverted List. *The 1st Conference on Graduate Research, SRRU*. Surin Rajabhat University, February, 2007, 104-115.

[20] wikipedia, (2006, 11, 15), *Hash table*. Available: http: http://en.wikipedia.org/wiki/Hash_table.

[21] K. Loudon, (2006, 11, 24), *Hash Tables*. Available: www.oreilly.com/catalog/masteralgoc/chapter/ch08.pdf.

[22] V. H. DINH, (2006, 11, 24), *Hash Table*. Available: http://libetpan.sourceforge.net/doc/API/API/x161.html.

[23] M. Escardo, (2006, 10,15), *Complexity considerations for hash tables* Avialable : http://www.cs.bham.ac.uk/ ~mhe/foundations2/node92.html.

[24] C. Khancome and V. Boonjing. Dynamic Dictionary Matching Using Inverted Lists. Proceeding of the Third IASTED International Conference ADVANCEDS IN COMPUTER SCIENCE AND TECHNOLOGY (ACST2007). April 2-4, 2007, Phuket,Thailand. 397-401.