An Efficient Composite-Alphabet Transform for String Matching under a Restricted Alphabet Set

Sung-Hwan Kim, Chang-Seok Ock and Hwan-Gue Cho
Pusan National University, Busan, South Korea
Email: {sunghwan, csock, hgcho}@pusan.ac.kr

***Abstract*— String matching is a problem of finding all occurrences of a short pattern on a relatively long reference string. While a number of methods have been presented, most published implementations assume several restrictions due to some practical issues. We focus on the restriction of the alphabet size, which is usually set to be 256 in many string matching libraries. When strings must be handled over an alphabet with a size greater than the limit provided by the given implementation, each character should be represented as a composite alphabet which involves combinations of two or more characters in the restricted alphabet set. In this case, potential false positives can sometimes occur, which may cause a decline in the performance in output-sensitive string matching systems, such as the FM-index. In this paper, we empirically compare various configurations of composite alphabets using FM-index, and show how they affect the performance in terms of the number of false positives and the searching time.**

***Index Terms*— Alphabet Size Limit, Composite Alphabet, FM-index, Output Sensitive Function, String Matching**

I. Introduction

String matching problems involve finding all occurrences of a given short-pattern query on a relatively long reference string. This problem has a number of applications, such as computer security [1], word recognition [2], data mining [3]-[5], and biological sequence processing [6]. Many theoretical solutions have been presented (including exact string matching algorithms [7], and approximate string matching algorithms [8]). A number of implementations for string matching and its applications have been published as software [9] or hardware [10].

In practice, however, there are several restrictions in the existing implementations due to practical issues such as code optimization or some limitations of programming languages. The alphabet size is one of the usual constrains that existing libraries have. Some implementations only focus on a specific domain with a very small alphabet set, such as biological sequences (the library may support only 'A', 'C', 'G', and 'T') [6]. In this case, the library usually uses several compression techniques so that it is very difficult to change the supported alphabet size. Another case is using the default character size that the language supports as the maximum alphabet size. According to the design, substantial changes in code to resize the alphabet may be involved. Even without these considerations, modifying the code is unsafe and has the potential to produce unreported errors.

To deal with this constraint of the alphabet size, there is a very straight-forward solution that can be applied without modifying the code, by representing the alphabet set as combinations of another alphabet set that is much smaller. This mapping is referred to as *alphabet transform*. As described in Figure 1, we can simply represent a large alphabet by compositions of characters from a smaller alphabet set.

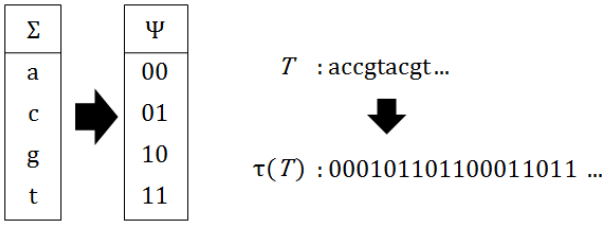

Figure 1. An example of an alphabet transform from {a,c,g,t} into {0,1}.

This approach, however, may produce substantial false positives. In Figure 2, every correct result should occur at the position $2i+1$ for some i, since we use a composite alphabet with length 3. However, we can find the query pattern at positions not in the form of $2i+1$. Such false positives must be filtered, and only positions of $2i+1$ must be extracted to obtain the result.

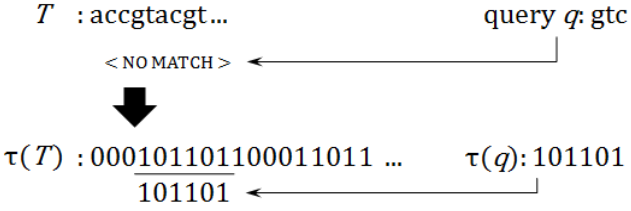

Figure 2. False positive problem. Although the original string does not contain the query, there are some search results from its transformed string.

Manuscript received October 20, 2012; revised November 15, 2012; accepted November 25, 2012

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No.2012-0005518). Corresponding should be addressed to Hwan-Gue Cho (hgcho@pusan.ac.kr).

doi:10.4304/jcp.8.7.1804-1809

Unfortunately, some useful string matching algorithms such as FM-index [11] involve costly operations to obtain the positional index of the search result. Moreover, they are usually sensitive to the number of occurrences of the given pattern on the indexed text. Thus false positives lead to increased search time.

Although using a long composite alphabet may reduce the number of false positives by increasing the composition space, longer queries and text that also affect the search performance are involved. In this paper, we analyze this trade-off in an empirical way. The search performance is experimentally compared with respect to various composite alphabet configurations.

II. Problem Definition

The string matching problem has a number of instances with respect to its given restrictions, and we discuss the following instance: given a long reference string $\boldsymbol{T}$, a self-index of $\boldsymbol{T}=t_1...t_n$ is constructed over Σ in advance, so as to find all occurrences i of a relatively short query string $\boldsymbol{q}=q_1...q_m$, which is given in run-time, such that $t_i...t_{i+m-1} = \boldsymbol{q}$ as fast as possible. But note that even if we focus on the exact string matching, the results of this paper are not only restricted to exact matching, but can be applied to any output-sensitive string matching framework, including approximate algorithms that allow some errors.

The alphabet transform examined in this paper, is the transform τ from Σ to Ψ^*, where Ψ is a smaller alphabet set, such that $|\Psi| \le |\Sigma|$. We focus on a particular case of the transform from Σ to Ψ^k for a fixed k. Although we can use Ψ to encode Σ by variable-length coding to reduce the space complexity, we consider only fixed-length coding for simplicity of the problem. Fixed-length coding supports direct access on any arbitrary element such that the corresponding position on the original string of the search results can be obtained easily.

III. Backgrounds: The FM-index

In this section, we review the FM-index, on which we mainly focus and conduct experiments in this paper. The FM-index [11] is a full-text indexing method that supports string matching using the Burrows-Wheeler transformation [12]. It additionally uses the wavelet tree for time and space efficiency [13]. Using the FM-index, reporting all *occ* occurrences of a pattern of length m on a reference string of length n where both strings are over alphabet Σ has the following time complexity [13]:

$$O((m + (occ \cdot \log^2 n) / \log\log n) \log|\Sigma|) \qquad (1)$$

The Burrows-Wheeler transform (BWT) is defined by a permutation of a string. The characters 'EOF' are first attached to the end of a given string. Conceptually, the BWT of the string is exactly the same as the last column of the matrix that is obtained by sorting all rotationally shifted strings of the EOF-attached string (see Figure 3). This process can be improved for computation in linear time using induced sorting [14].

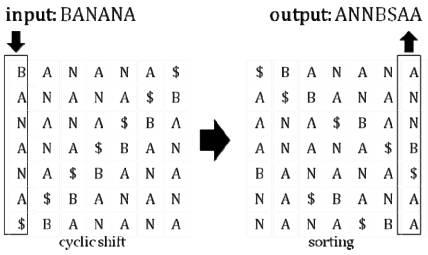

Figure 3. The Burrows-Wheeler transform of string "BANANA."

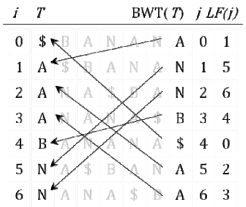

Figure 4. LF-mapping for the BWT of "BANANA."

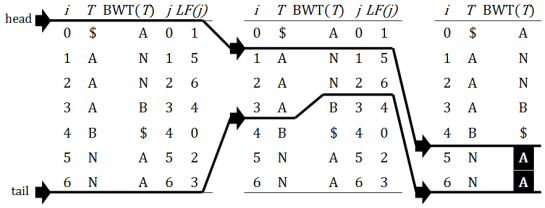

Figure 5. Searching process of "NA". The result characters "A" of all "NA"s are indicated by black boxes.

The search process of the BWT is based on LF mapping [12], which connects identical characters from the last column of the matrix of sorted cyclic-shifted strings to the first column (see Figure 4).

Since the first column of the matrix contains the precedent characters of those in the last column, LF-mapping has a property that the number of occurrences of α in the first i rows of the last column is the same as that in the first LF(i) rows of the first column. String matching queries can be processed using double pointers and LF-mapping as described in Figure 5. Starting from pointers at the top and bottom, we find the first and last occurrence of the current character between two pointers, then reset pointers to the positions of the LF-mapping points. However, finding the occurrences of a character α is costly and involves substantial computation or memory resources. Additionally, after processing the final character, we know only the number of occurrences of the given pattern. If we want to obtain exactly where they are, and we have to track each resulting output to derive its location on the original string, which is another bottleneck in processing search queries, and this is why false positives cause problems.

To deal with this problem, the FM-index uses the BWT with additional techniques such as suffix array sampling and the wavelet tree to boost the search performance without sacrificing space complexity. This adjusts the trade-offs between the space and time complexity very well.

The wavelet tree [15] is an efficient data structure for **rank** and **select** queries, which are used to obtain the occurrences discussed above and defined on binary vectors as follows: a **rank** query (α,i) on string T asks to find the number of occurrences of α on the prefix $T[1..i]$, and a **select** query (α,i) asks the position of the ith α on the whole string T. The wavelet tree uses less space to store the given string T as a binary tree whose nodes are represented as bit vectors so that each query is processed in $O(\log|\Sigma|)$ time, where Σ is the alphabet over which strings are defined (see Figure 6).

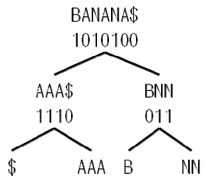

Figure 6. An example of the wavelet tree of string "BANANA$."

IV. Transform into Composite Alphabets

When we use an implementation of the string matching algorithm that has the limitation on the alphabet size L, while strings to be handled are defined over an alphabet Σ of a larger size than that the library supports, we have to represent the strings as combinations of k characters in an alphabet set Ψ that is sufficiently small. This mapping from Σ into Ψ^k is denoted by τ.

A straight-forward approach involves choosing any alphabet Ψ of a size smaller than L, and choosing k to be larger than $|\Psi|^k \geq|\Sigma|$. Then, we associate each element in Ψ^k with each of Σ. Figure 7 shows an example in which Σ is the English alphabet {a,...,z} and Ψ={a,c,g,t}. To represent all of the letters in the alphabet, at least 4 characters in Ψ should be combined. We use k=4 here, so each composite alphabet in Ψ^k is a string of length 4. We assign them for each of the English letters in lexicographical order. As discussed above, we can easily find an example of false positives. In Figure 7, applying the inverse transform of the shifted strings of the transformed string over the composite alphabet, we can obtain false positive matching results such as "qdqdq," which is not in the original string.

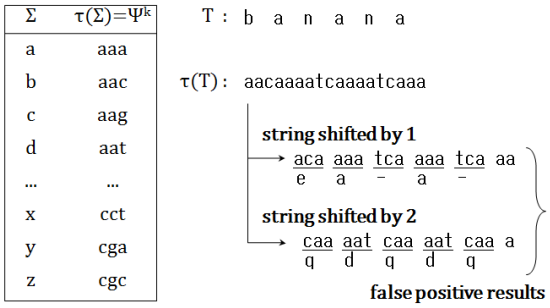

Figure 7. An example of composite alphabet transform. Searching on the transformed string may involve unnecessary results that are not reported in the original string.

Given an alphabet transform τ: $\Sigma\rightarrow\Psi^k$, we can denote $N_\tau(n,m)$ to be the expected number of false positives for n, m which are the lengths of the indexed string and query string, respectively. Our objective is to find τ that leads to good performance. Intuitively, reducing $N_\tau(n,m)$ is necessary to do so.

If $\Sigma=\Psi^k$ and all characters in Σ appear in a uniformly random manner -- that is, $p(\alpha)=p(\beta)=1/|\Sigma|$ for all α, $\beta\in\Sigma$ -- then we have $N_\tau(n,m)=(k\text{-}1)(n\text{-}m)/|\Sigma|^m$, since, we have k-1 chances with the probability of $1/|\Sigma|^m$ for each n-m possible positions for the appearance of false positives. Figure 8 shows the expected number of false positives with respect to n and m under this configuration.

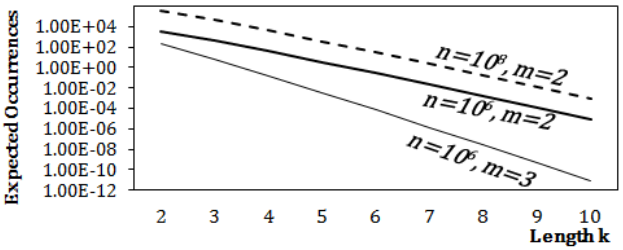

Figure 8. The expected number of the occurrences of false positive results with several configurations of m, n, with respect to composite alphabet length k.

If there are false positives on strings over a composite alphabet obtained by alphabet transform τ, it is implied that there are some characters α, $\beta\in\Sigma$ such that the string represented as the concatenation of the transformed composite characters $\tau(\alpha)\tau(\beta)$ has some substring $x=\tau(\gamma)$ that is neither a prefix nor suffix for some $\gamma\in\Sigma$. For example, under the configuration τ with $\Sigma=\Psi^k$ we discussed above, for any α and β, every k-length substring of $\tau(\alpha)\tau(\beta)$ is $\tau(\gamma)$ for some $\gamma\in\Sigma$, such that substantial false positives are produced.

If we use a specific subset of $\Psi^{k'}$ with a larger k' rather than all elements in Ψ^k with the smallest k for representing elements of Σ, we can drastically reduce the number of false positives. For example, using τ, we can construct a transform τ' that raises no false positives, by (i) choosing $\alpha\in\Psi$ as a prefix character, and (ii) setting k to be the smallest integer such that $(|\Psi|\text{-}1)^k \geq|\Sigma|$, then (iii) associating each $\beta\in\Sigma$ with $\tau'(\beta)=\alpha\cdot\tau(\beta)$. Since the delimiter α does not appear on $\tau'(\beta)$ except as its heading character, we can have $N_{\tau'}(n,m)=0$ in this configuration.

As discussed in previous sections, the searching time depends on both the number of occurrences and the length of the query. We can reduce the number of occurrences by choosing a good alphabet transform τ, but this can lead to increased query length. Figure 9 shows this trade-off between the number of false positives and the length of transformed strings.

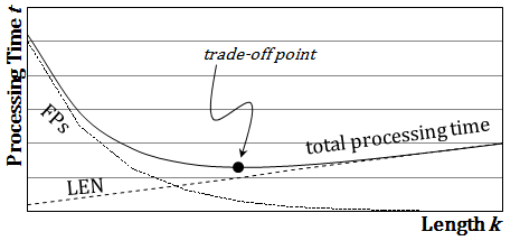

Figure 9. Trade-off between false positives (FPs) and transformed string lengths (LEN) with respect to the length k of the composite alphabet.

Note that the former case of $\Sigma=\Psi^k$ has the minimum k but produces the largest number of false positive results, and the latter case that uses a specific character as a prefix delimiter involves the minimum (zero) false positives, but has a longer composite alphabet. No consideration for composite alphabets longer than that of this case is necessary, since we have a sufficient setting to eliminate all false positives. From these two extreme cases, we obtain the following lower and upper bounds of an optimal k for a fixed Ψ:

$$\frac{\log|\Sigma|}{\log|\Psi|} \le k \le 1+\frac{\log|\Sigma|}{\log(|\Psi|-1)} \qquad (2)$$

V. The Incremental Selection Algorithm

In this section, we present an algorithm that selects a subset of Ψ^k for mapping incrementally. As discussed in the previous section, false positive results are caused by selecting strings for which some concatenations have a string $x=\tau(\gamma)$ as their substrings for some $\gamma\in\Sigma$. Focusing on this observation, we can construct an incremental selection strategy.

The algorithm is shown in Figure 10. We use two sets for storing prefixes and suffixes of selected strings to check whether the next given string can be chosen without the possibility of raising false positive results. Iterating all possible strings, we exclude strings that have a period shorter than itself, or can be constructed by concatenating a suffix and a prefix of strings that are already selected. If a string is acceptable to be chosen, we insert it into the resulting set and insert all proper prefixes and proper suffixes for checking the next strings.

This algorithm gives a composite alphabet set of length k from Ψ that does not produce any false positives, since the strings that produce unnecessary results are filtered out by checking prefix and suffix conditions. The resulting alphabet set can represent more characters than the method using a delimiter character to suppress false positive results.

VI. Comparative Evaluation

In this section, we have conducted some experiments to show a comparative analysis of different composite alphabet configurations. All experiments are run on a workstation equipped with a 3.33-GHz Intel Core i5 CPU and 4 GB of RAM. We used fm-index++ 0.0.5[1] as the implementation of the FM-index.

For the experiments, we generated a random string of 10-MB length over an alphabet of size 64. Composite alphabet transforms based on two introduced straight-forward methods: one denoted as CPT which uses the full space obtained by the power of a given target alphabet to represent the alphabet where strings are defined; and the other one, denoted as DLM, uses a specific character to distinguish each character to suppress the occurrences of false positive results. Our proposed method of incremental selection is referred to as INC.

Figure 11 shows the relative false positives defined by the ratio of the number of false positives to the number of true positives (the number of occurrences before transformation). When the query length is larger than 4, the effect of false positives becomes small enough to ignore.

Algorithm	*Select_String_Samples*	
Input	Ψ	Set of composite alphabet letters.
	k	Length of composite alphabet.
Output	S ($\subset \Psi^k$)	Resulting composite alphabet.

```
Procedure
  S←ϕ
      S_prefix ← ϕ
  S_suffix ← ϕ
  For each s ∈ Ψ^k
    If ∃u∈Ψ* s.t. u^n=s Then Next s
    prefix_check ← false
    suffix_check ← false
    For i = 1 to k-1
      If s[k-i...k-1] ∈ S_prefix Then prefix_check←true
      If s[1...i] ∈ S_suffix Then suffix_check←true
      If ( prefix_check and suffix_check ) or
         ( prefix_check and s[1...k-i] = s[i...k-1] ) or
         ( suffix_check and s[1...i] = s[k-i...k-1] ) Then
           Next s
      End If
    End For
    S←S∪{s}
    For i = 1 to k-1
      S_prefix←S_prefix ∪ {s[1...i]}
      S_suffix←S_suffix ∪ {s[k-i...k-1]}
  End For
  Return S
End Procedure
```

Figure 10. Algorithm for string sampling to obtain a better transform.

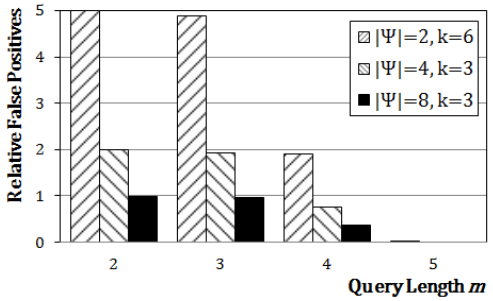

Figure 11. Relative false positives of CPT method with respect to target alphabet size, composite alphabet length and query length.

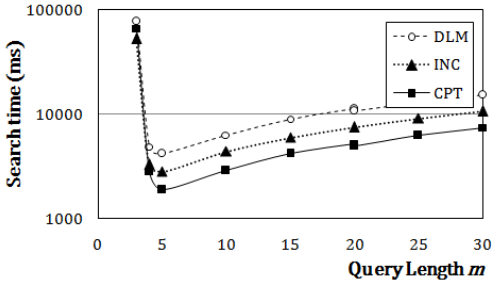

Figure 12. Search performance with respect to query length. CPT method is fastest except when m=3, where the proposed method (INC) is the fastest.

[1] Available at http://code.google.com/p/fmindex-plus-plus.

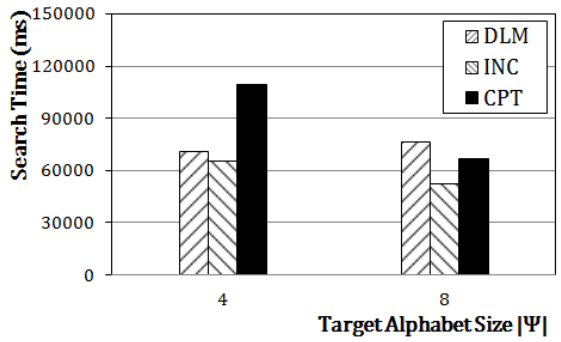

Figure 13. Search performance comparison when the query length m=3. The proposed method (INC) performs the best.

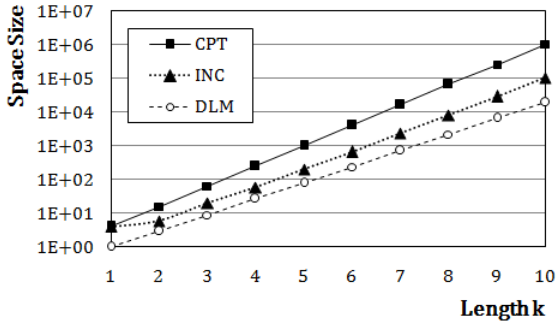

Figure 14. Size of generated string space with respect to composite alphabet size. The proposed method provides an acceptable size of composite alphabet space without raising any false positives.

Fig. 13 describes the search time with respect to |Ψ| for the query length m=3 where the number of occurrences of the pattern is not small enough to ignore, and false positives are likely to appear. The proposed method (INC) outperforms the other methods in this environment.

Fig. 14 shows the space size generated by each method. DLM has the smallest space since it uses only one character for denoting the character boundary. CPT uses all possible combinations of the given target alphabet, but it produces a number of false positives as we have noted. The proposed method effectively adjusts the trade-off between these two extreme methods and provides an acceptable size of the alphabet space without producing any false positives.

VII. Conclusion

We have discussed the practical aspects of using string matching tools, particularly focusing on the limit of the alphabet size by the composite alphabet transformation. Our contribution is summarized as follows:

- We introduce the problem of the composite alphabet transform and discuss the trade-off in constructing this mapping.
- We also presented an incremental selection algorithm, which effectively adjusts this trade-off, and provides the best performance when the query length is short enough such that the occurrences of false positives cannot be ignored.

Although we have demonstrated the performance of our proposed method, more extensive experiments on datasets with various characteristics and theoretical analysis should be addressed in future work.

References

[1] I. Balasundaram and E. Ramaraj, "An Efficient Technique for Detection and Prevention of SQL Injection Attack using ASCII Based String Matching," *Procedia Engineering*, vol. 30, pp. 183-190, 2012.

[2] L. Guerrouj, M. Di Penta, Y. Gueheneuc and G. Antoniol, "Rocignizing Words from Source Code Identifiers Using Speech Recognition Techniques," in *Proc. 14th European Conf. on Software Maintenance and Reengineering*, pp. 68-77, 2010.

[3] S. W. K. Chan, "Shallow Semantic Labeling Using Two-phase Feature-enhanced String Matching," *Expert Systems with Applications*, vol. 36, pp. 9729-9736, August 2009.

[4] P.-C Huang, S.-S. Lee, Y.-H Kuo and K.-R. Lee, "A Flexible Sequence Alignment Approach on Pattern Mining and Matching for Human Activity Recognition," *Expert Systems with Applications*, vol. 37, pp.298-306, January, 2010.

[5] T. P. Exarchos, M. G. Tsipouras, C. Papaloukas and D. I. Fotiadis, "A Two-stage Methodology for Sequence Classification Based on Sequence Pattern Mining and Optimization," *Data & Knowledge Engineering*, vol. 66, pp. 467-487, September 2008.

[6] H. Li and R. Durbin, "Fast and Accurate Short Read Alignment with Burrows-Wheeler Transform," *Bioinformatics*, vol. 25, no. 14, May 2009.

[7] C. Charras and T. Lecroq, *Handbook of Exact String-Matching Algorithms*, College Publications, 2004.

[8] G. Navarro, "A Guided Tour to Approximate String Matching," *ACM Computing Surveys*, vol. 33, no. 1, pp. 31-88, March 2001.

[9] G. Navarro, "NR-grep: A Fast and Flexible Pattern-matching Tool," *Software: Practice and Experience*, vol. 31, no. 13, pp. 1265-1312, November 2001.

[10] Q. Li, J. Li, J. Wang, B. Zhao and Y. Qu, " A Pipelined Processor Architecture for Regular Expression String Matching," Micro- processors and Microsystems, vol. 36, no. 6, pp. 520-526, August 2012.

[11] P. Ferragina and G. Manzini, "Opportunistic Data Structures with Applications," in *Proc. 41st Annual Symposium on Foundations of Computer Science*, pp. 390-398, 2000.

[12] M. Burrows and D. Wheeler, "A Block Sorting Lossless Data Compression Algorithm," *Technical Report 124, Digital Equipment Corporation*, 1994.

[13] P. Ferragina, G. Manzini, V. Mäkinen and G. Navarro, "An Alphabet-Friendly FM-index," in *Proc. 11th Symposium on String Processing and Information Retrieval*, pp. 150-160, 2004.

[14] D. Okanohara and K. Sadakane, "A Linear-Time Burrows-Wheeler Transform Using Induced Sorting," in *Proc. 16th Symposium on String Processing and Information Retrieval,* pp. 90-101, 2009.

[15] R. Grossi, A. Gupta and J. S. Vitter, "High-order Entropy-compressed Text Indexes," in *Proc. 14th Annual SIAM/ACM Symposium on Discrete Algorithms*, pp. 841-850, 2003.

Sung-Hwan Kim is a M.S student in Pusan National University. He received the B.S. degree from Pusan National University. His research interests are information retrieval and sequence processing.

Chang-Seok Ock is a M.S. student in Pusan National University. He received the B.S. degree from Pusan National University. His research interests are information retrieval, human-computer interaction, and computer graphics.

Hwan-Gue Cho is a Professor in Pusan National University. He received the B.S. degree from Seoul National University, Korea, and the M.S and Ph.D. degrees from Korea Advanced Institute of Science and Technology, Korea. His research interests are computer algorithms and bioinformatics.