String Filtering of a Large String Collection on Mobile Devices using a Neural Network

Chia-Cheng Liu[12] Heng Ma[2]

[1] Department of Industrial Management
Vanung University
Chung-Li, Taoyuan 320,Taiwan.
[2] Department of Industrial Management
Chung-Hua University
Hsinchu 300, Taiwan.

Abstract

String matching of a large string collection on mobile devices has been a difficult problem because of the memory space and computing speed constraints. We propose a method to efficiently determine whether a query string exists in the large string collection. The proposed method, based on a string encoder using a neural network, can perform the task in a consistent manner regardlss of the number of strings in the collection. The string encoding process is executed on the server site, and the result is transmitted to the PDAs via the Internet. The server checks a remote database containing the most-updated data constantly, and thus renders the PDAs the accuracy in filtering the query strings. We conducted an experiment that simulates the stolen car license plate matching, and the results were favorable in both speed and memory space requirement.

Keywords: Text String, Mobile Devices, Neural Network

1. Introduction

Mobile devices, with their portability, have becoming an important tool for a number of applications, such as detection of certain strings and pictures, and on-line information retrieval in the outdoors. In this paper, we address a method that can transform a large collection of strings, such as stolen car license plates, into a feasible size, and the computation complexity for determining whether a given string is within the large collection is approximately $O(1)$ regardless of the size of the collection. The method is particularly suitable for applications requiring constant checking of an encountering string, and the existential status of the string must be determined in a timely fashion. In a sense, the problem is a special type of the exact string matching problem. A great number of algorithms have been developed for solving the exact string matching problem since nearly three decades ago, such as Aho-Corasick [1], Boyer-Moore [2] and Knuth-Morris-Pratt [3]. A general purpose of these algorithms is to find all the locations or to calculate the occurrence of the string pattern in the target text, in which the computational complexity must be as little as possible. More recently, some indexing methods, e.g. the inverted files [4], the B-tree [5] and suffix arrays [6][7], have devoted themselves to reducing the computational complexity by constructing indexing structures for accelerating the querying speed. The extra memory required by these indexing structures, however, presented a drawback because they are difficult to be implemented on the primary memory when the size of the target text becomes extremely large. Although some researchers [8][9], in order to relieve the overhead on the primary memory, have developed schemes to store the extra memory required by the indexing structures on the external memory space, retrieving the indices from the external memory space usually result in degradation of the overall efficiency, which is critical for real-time applications. To resolve the space-efficiency dilemma, some other methods have been proposed, in which text or index pre-processing means were developed. These methods, including a hybrid sorting algorithm [10], augmented suffix arrays [11], compression [12][13][14], and coding techniques, such as Huffman coding [15] and Tagged Sub-optimal Code [16], have made great efforts for reducing the space-efficiency barrier.

Recently, as the Internet prevails, Intrusion Detection Systems (IDSs) have been recognized as powerful tools for the detection of malicious attacks over the network [17][18]. Such systems usually contain a large key-string collection composed of a number of rules, mostly generated by experts, for

identifying a variety of known attacks, where an exact string matching mechanism is essential. Therefore, time and space efficiency is a fundamental requirement for such a mechanism since the IDS detects the attacks by searching through packets in run time and identifying the contents that match the rules in the collection. The data structure of the string collection or the payload could be varied for different systems as long as it meets both the time and space requirements. The construction time for the data structure is usually excluded from the run-time performance, and a technique with a consistent performance independent of the size of the payload is very desirable [18]. The proposed method is a promising tool for such a task. The payload in our method, however, is a huge list of strings instead of rules. The strings in the payload are supposed to be known *a priori* and should be frequently updated.

For the updating purpose, the proposed method is designed to be implemented using the client-server architecture on the Internet as shown in Fig. 1. For the server site implementation, the string encoder is developed using a neural network to transform the large string collection into an encoded file, which is composed of a one-dimensional array of numerical values. A certain degree of compression could be achieved after the transformation process because of the hashing characteristic between the input and the intermediate layers in the employed neural network. The mapping process is performed in the off-line mode. After the mapping process is completed, the encoded file is generated and stored in an area where the file transmission service is provided. For the client site implementation, the PDA, the on-line component embedded in a host application will download the encoded file from the server site whenever the file is updated. When a query string is input to the PDA, the component only refers to a few cells of the one-dimensional array in the encoded file for determining the existential status of the query string, and thus renders itself a consistent speed regardless of the size of the payload.

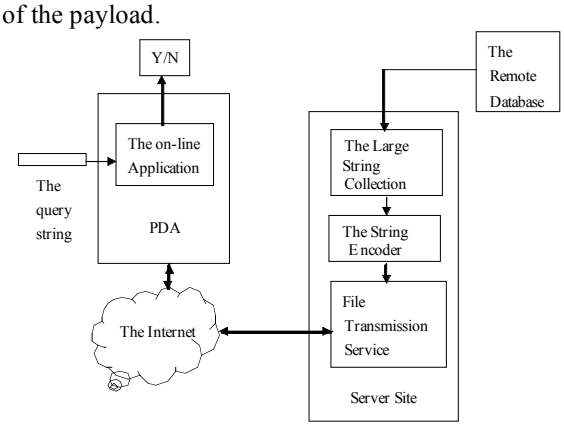


Fig. 1: The schematic diagram of the proposed method

2. The string encoder using a neural network

The neural network employed in this paper is the CMAC (Cerebellar Model Articulation Controller) proposed by Albus in 1975 [19][20], and was originally designed for controlling a robotic manipulator with multiple degrees of freedom. As shown in Fig. 2, Albus employed a one-dimensional array including a number of cells, each of which is associated with a weight value for the mapping between the input and the output vectors.

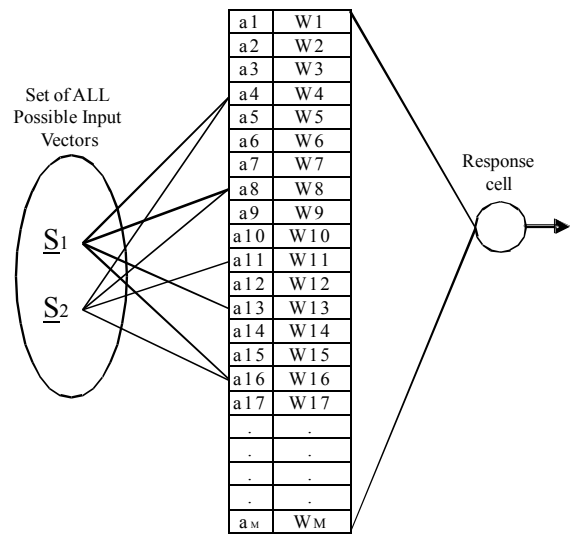


Fig. 2: The architecture of Albus's CMAC

The one-dimensional array, referred to as the association layer, acts as an intermediate mapping area between the input space and the output node. Each input vector activates a small number of the cells in the association layer, whose weight values are then summed into the output signal. The weight values in the association layer are initially set to zeros and consequently converge to specific values according to a simple weight modification rule. The rule can be realized by simply calculating the difference between the output signal and the desired signal, also referred to as the error, and then propagates a fraction of the error to each of the activated cells. In a sense, all the input vectors are hash-coded into the association layer in an iterative fashion until the output signal is converged within specified thresholds. The weight modification process of CMAC renders itself a good capability of memorization and a quick convergence speed, which are very suitable for our implementation as a text string encoder.

For the implementation of our string encoder using CMAC, generalization is not an important factor

since our goal, in a sense, is to memorize all the strings in the string collection. For the purpose of memorization, a mapping function that can uniformly activate the cells in the association layer by the input vectors is more desirable. Ellison [21] pointed out that the logarithm of a number discarding the first few digits is more suitable for such a task. We adopt Ellison's work as our mapping function between the input space and the association layer. Furthermore, in our implementation, the number of activated cells is identical to the string length, i.e., each character of the input string activates a cell in the association layer. The mapping function for our string encoder has the form as in (1), and the output signal produced by the input string s is represented by $O(s)$, which is obtained by summing the weight values of the activated cells as in (2).

$$f(s(i)) = Int(\log_{base}(\sum_{j=1}^{i} Ascii(s(j))) \cdot \hat{E}) \bmod M \qquad i = 1 \sim l(s) \tag{1}$$

$$O(s) = \sum_{i=1}^{l(s)} w(f(s(i))) \tag{2}$$

where $f(s(i))$ represents the position of the cell activated by the ith character of the input string s, $Ascii(s(j))$ is the decimal ASCII code of the character $s(j)$, K is a large number, M is the size of the association layer and $l(s)$ is the length of s.

In (1), $f(s(i))$ has the integer value between zero and M-1. The summation operation inside the logarithmic function is utilized to intensify the character order of the input string. Therefore, strings with more characters in an identical order from the starting character will activate more cells in common, which could alleviate the mapping overhead of the association layer. K is a constant factor for avoiding the situation where only a portion of the association layer is activated by all the input vectors. In a sense, the range of the activated cells is amplified by K, and ideally the range must cover the entire association layer. Therefore, we select K as the smallest prime number exceeding M to avoid such a situation. Intuitively, the size of the association layer, representing the memory space required to store the weight values, could affect the convergence speed, i.e. the larger the association layer, the shorter the convergence time. This is because the number of activation of a cell by all the strings in the payload decreases as the size of the association layer increases. However, a large association layer will occupy more memory space to store the weight values. Therefore, a reasonable size of the association layer must be determined in order to keep both the mapping time and the storage memory space at an acceptable level.

The CMAC mapping belongs to the supervised type, in which a target value representing the desired output signal for each input vector must be assigned. It is conceivable that the target values for all input vectors should be uniformly located within a small range centered at zero so that the mapping process can converge smoothly [21]. Secondly, the target value must be a function of the input string for our implementation since there is no designated target value for each string in the payload. Thirdly, the target value should not be the same for all the strings in the payload because the mapping process will degenerate when all the strings are in an identical length. This is because all the weight values in the association layer will converge to the same one, which is the target value divided by the identical string length. Therefore, we take advantage of the uniformity of $f(s(i))$ in (1), and the target value function $T(s)$ for our implementation has the form as in (3).

$$T(s) = \begin{cases} +1 & if\ (\sum_{i=1}^{l(s)} f(s(i)) / l(s)) > (M/2) \\ -1 & otherwise \end{cases} \tag{3}$$

Once $T(s)$ is decided, the weight modification process is to propagate the error multiplied by a learning rate to the activated cells for each input string as in (4).

$$w(i)_{new} = w(i)_{old} + \eta \cdot (T(s) - O(s)) \tag{4}$$

for all i activated by s

where $w(i)_{new}$ and $w(i)_{old}$ are the new and old weights at the ith cell respectively, η is the learning rate.

As the mapping process proceeds, the output signal produced by the input string will gradually move toward to the corresponding target value. After the mapping process is completed, the error range that includes all the strings with the same target value is calculated as in (5) and (6). This is accomplished by determining the upper and lower bounds of error associated with each target value. We refer to the error range as the acceptance window because a query string is considered as that it exists in the payload if the corresponding output signal is inside the window.

$$U(t_i) = Max(O(s_j) - t_i) \quad \forall\, j, T(s_j) = t_i \tag{5}$$

$$L(t_i) = Min(O(s_j) - t_i) \quad \forall\, j, T(s_j) = t_i \tag{6}$$

where $U(t_i)$ and $L(t_i)$ are the upper and lower bounds of error range for the ith target value and $O(s_j)$ is the output signal for the jth string in the payload.

The purpose of the mapping process is to minimize the enclosing range of the errors for each target value because a smaller range results in a higher accuracy in determining the existential status of a query string. When all the error ranges do not significantly reduce in a number of iterations, e.g. the reduction is smaller than 0.001 in 200 iterations for all target values, the mapping process terminates and the

target values with their corresponding upper and lower bounds are recorded in the encoded file.

3. The PDA implementation

The on-line component on the PDA plays the role as a core engine for determining the existential status of a query string, and is usually embedded in an application that must constantly perform the existence-checking task for a query string. Therefore, the component must be robust in efficiency so that the performance of the host application is not affected. Furthermore, the memory requirement of the component must be as small as possible since it occupies the primary memory in run time. The kernel of the component is mostly the same as the string encoder except that the error of a query string s is not propagated to the association layer. The error, however, is utilized to determine the existential status of s. When the error is within the error range of the corresponding target value, a *true* response is submitted to the host application; otherwise, a *false* response is given as in (7).

$$E(s) = \begin{cases} true & if\ L(T(s)) \le O(s) - T(s) \le U(T(s)) \\ false & otherwise \end{cases} \quad (7)$$

where $E(s)$ represents the existential status of string s, $O(s)$and $T(s)$ are the output signal and the target value of s respectively, and $L(T(s))$ and $U(T(s))$ are the lower and upper bounds of the error range associated with the target value of s.

4. Results

The proposed method was implemented using the C language, and the simulated scenario was to identify stolen license plates on a PDA. The server utilized was a PC equipped with an Intel P4 CPU at 3.0GHz and 1GB of DDR SDRAM, and the PDA was a Unitech PA960 based on an Intel ARM-SA1110 at 206MHz with 64MB RAM. The license plates were strings in a fixed length of seven, in which the separation mark, a dot, was at the middle position. Fig. 3 shows some of the sample license plates.

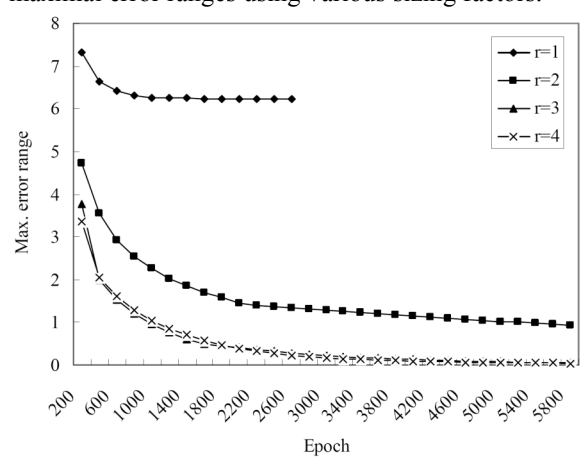
0K6せTLH
CWTせQ6U
PIAせ03D
55Zせ40
0DJせKLL
5NGせC2L
SSIせAVQ

Fig. 3: Sample license plates for the experiment

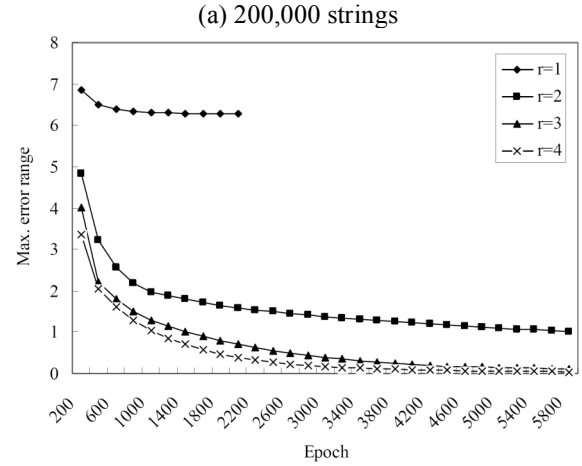
We randomly generated 500,000 sample license plates, 400,000 of which were utilized as the training set, and the remaining 100,000 plates were used as the testing set. Since the size of the association layer is a crucial factor in the mapping process, we conducted several mappings using different sizes of the association layer. In each of these mappings, the size of the association layer was determined by calculating a specific fraction of the memory space required by the plain-text string collection. Let n be the memory requirement in bytes of the training set, and m be the bit size of the association layer. We characterized a sizing factor r representing the relationship between m and n as shown in (8).

$$m = r \cdot n \quad (8)$$

It should be noted that the memory requirement of the association layer is equal to that of the plain-text training set when r=8. We enumerated r from one to four with an increment of one for the mapping purpose. Fig. 4 shows the relationships between the number of iterations (epochs) in the mapping process and the maximal error ranges using various sizing factors.



(a) 200,000 strings



(b) 400,000 strings

Fig. 4: The mapping results for the simulated scenario using various sizing factors

As shown in Fig. 4, it can be realized that as the sizing factor increased, the maximal error range converged in a relatively shorter time. Table 1 provides the details of the mapping process for the simulated scenario.

Table 1: The mapping details for the simulated scenario

	Sizing factor (r)	Number of cells in the association layer	Number of epochs elapsed	Mapping time on the designated PC (seconds)	Maximal error range after mapping
200,000 strings	1	109,875	2,600	189	6.170492
	2	219,625	46,000	3,658	0.124482
	3	329,625	19,000	2,057	0.020978
	4	439,500	8,800	1,187	0.006928
400,000 strings	1	219,750	2,000	295	6.255294
	2	439,500	36,600	6,323	0.212063
	3	659,250	12,400	3,076	0.017560
	4	879,000	9,000	2,661	0.009934

The purpose of testing was to evaluate the accuracy and the efficiency of the on-line component while determining the existential status of a query string in real time. There are two types of error in determining the existential status of a query string, namely Type I and Type II errors. The Type I error refers to as the situation where a query string is determined as *false* when it actually exists in the training set, and the Type II error is the opposite. In the proposed method, however, only the Type II error is possible, i.e. the false-alarm rate (FAR), since each string in the training set is enclosed by the corresponding error range after the mapping process. Therefore, we only evaluated the FAR in each mapping. In addition, since the computational efficiency is critical for the on-line component, the averaged times for processing a query string were also presented. Table 2 shows the testing results.

Table 2: The testing results for the simulated scenario

	Sizing factor (r)	FAR (%) of 100,000 testing strings	Averaged time for processing a query string
200,000 strings	1	96.906	1.2 ms
	2	1.823	1.1 ms
	3	0.24	1.2 ms
	4	0.17	1.2 ms
400,000 strings	1	98.868	1.1 ms
	2	4.339	1.3 ms
	3	0.503	1.2 ms
	4	0.33	1.1 ms

Table 2 conveys two pieces of information: (a) the FAR tends to decrease as the size of the association layer increases, and (b) the averaged time for processing a query string by the on-line component is very consistent regardless of the number of strings in the training set.

5. Conclusion

In this paper, we present a string-filtering method based on a string encoder using the CMAC. The proposed method performs in an efficient manner for determining whether a string exists in a given large string dataset, which is suitable for real-time applications. The advantages of the proposed method include: (a) the computational time for processing a query string is nearly a constant regardless of the amount of strings in the dataset, (b) the memory requirement of the on-line querying component is relatively light, and is therefore suitable for the implementations on mobile devices, and (c) the mapping process is quite efficient, and the encoded file can be transmitted on the Internet in a matter of seconds, which render a timely and up-to-date status for the encoded files on the PDA. Although the proposed method inevitably accompanies with a small amount of Type II errors, the odd of occurrence is relatively small in the addressed application

References

[1] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, pp. 333-340, 1975.

[2] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 761-772, 1977.

[3] D. Knuth, J. Morris, and V. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, vol. 6, pp. 323-350, 1977.

[4] A. Moffat and J, Zobel, "Self-Indexing Inverted Files for Fast Txt Retrieval," *ACM Transactions on Information Systems*, vol. 14, no. 4, pp. 349-379, 1996.

[5] P. Ferragina and R. Grossi, "The String B-tree: a New Structure for String Search in External Memory and its Applications," *Journal of ACM*, vol. 46, no. 2, pp. 236-280, 1999.

[6] U. Manber and E. W. Myers, "Suffix Arrays: A New Method for On-Line String Searches," *SIAM Journal on Computing*, vol. 22, no. 5, pp. 935-948, 1993.

[7] M. I. Abouelhoda, E. Ohlebusch and S. Hurtz, "Optimal Exact String Matching Based on Suffix arrays," *Proceedings of the Ninth International Symposium on String Processing and Information Retrieval*, 2002.

[8] J. S. Vitter, "External Memory Algorithms and

Data Structures: Dealing with Massive Data," *ACM Computing Surveys*, vol. 33, no. 2, pp. 209–271, 2001

[9] A. Crauser and P. Ferragina, "A Theoretical and Experimental Study on the Construction of Suffix Arrays in External Memory," *Algorithmica*, vol. 32, pp. 1-35, 2002.

[10] J. Bentley and R. Sedgewick, "Fast Algorithms for Sorting and Searching Strings," *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pp. 360-369, 1997.

[11] L. Colussi and A. De Col, "A Time and Space Efficient Data Structure for String Searching on Large Texts," Information Processing Letters, vol. 58, no. 5, pp. 217-222, 1996.

[12] E. S. De Moura, G. Navarro, N. Ziviani and R. Baeza-Yates, "Fast and Flexible Word Searching on Compressed Text," *ACM Transactions on Information Systems*, vol. 18, no. 2, pp. 113-139, 2000.

[13] R. Grossi and J. S. Vitter, "Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching," *SIAM Journal on Computing*, vol. 35, no. 2, 2005.

[14] N. Ziviani, E. de Moura, G. Navarro and R. Baeza-Yates, "Compression: A Key for Next-Generation Text Retrieval Systems," *IEEE Computer*, vol. 33, no. 11, pp. 37-44, November, 2000.

[15] S. Pigeon and Y. Bengio, "A Memory-Efficient Adaptive Huffman Coding Algoritm for Very Large Sets of Symbols," *Data Compression Conference*, 1998.

[16] A. Bellaachia and I. AL Rassan, "Speeding up String Matching over Compressed Text on Handheld Devices using Tagged Sub-optimal Code," *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2004.

[17] S. Antonatos, K. G. Anagnostakis, and E. P. Markatos, "Generating Realistic Workloads for Network Intrusion Detection Systems," *ACM Workshop on Software and Performance*, 2004.

[18] N. Tuck, T. Sherwood, B. Calder and G. Varghese, "Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection," *Proceedings of the IEEE Infocom Conference*, Hong Kong, China, March 2004.

[19] J. S. Albus, "A New Approach to Manipulator Control: The Cerebellar Model Articulation Controller (CMAC)," *Journal of Dynamic Systems, Measurement and Control*, vol. 97, no. 3, pp. 220-227, 1975.

[20] J. S. Albus, "Data Storage in the Cerebellar Model Articulation Controller (CMAC)," *Journal of Dynamic Systems, Measurement and Control*, vol. 97, no. 3, pp. 228-233, 1975.

[21] D. Ellison, "On the Convergence of the Multidimensional Albus Perceptron," *The International Journal of Robotics Research*, vol. 10, no. 4, pp. 338-357, 1991.