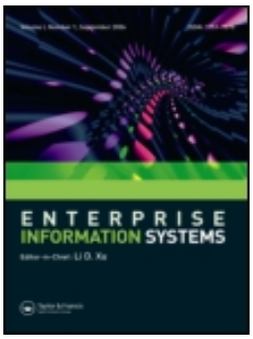

Enterprise Information Systems

ISSN: 1751-7575 (Print) 1751-7583 (Online) Journal homepage: https://www.tandfonline.com/loi/teis20

An enhanced dynamic hash TRIE algorithm for lexicon search

Lai Yang , Lida Xu & Zhongzhi Shi

To cite this article: Lai Yang , Lida Xu & Zhongzhi Shi (2012) An enhanced dynamic hash TRIE algorithm for lexicon search, Enterprise Information Systems, 6:4, 419-432, DOI: 10.1080/17517575.2012.665483

To link to this article: https://doi.org/10.1080/17517575.2012.665483

Enterprise Information Systems
Vol. 6, No. 4, November 2012, 419–432

Taylor & Francis
Taylor & Francis Group

An enhanced dynamic hash TRIE algorithm for lexicon search

Lai Yang[a]*, Lida Xu[a,b] and Zhongzhi Shi[a]

[a]*Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China;* [b]*Old Dominion University, Norfolk, VA 23529, USA*

(*Received 20 January 2011; final version received 7 February 2012*)

Information retrieval (IR) is essential to enterprise systems along with growing orders, customers and materials. In this article, an enhanced dynamic hash TRIE (eDH-TRIE) algorithm is proposed that can be used in a lexicon search in Chinese, Japanese and Korean (CJK) segmentation and in URL identification. In particular, the eDH-TRIE algorithm is suitable for Unicode retrieval. The Auto-Array algorithm and Hash-Array algorithm are proposed to handle the auxiliary memory allocation; the former changes its size on demand without redundant restructuring, and the latter replaces linked lists with arrays, saving the overhead of memory. Comparative experiments show that the Auto-Array algorithm and Hash-Array algorithm have better spatial performance; they can be used in a multitude of situations. The eDH-TRIE is evaluated for both speed and storage and compared with the naïve DH-TRIE algorithms. The experiments show that the eDH-TRIE algorithm performs better. These algorithms reduce memory overheads and speed up IR.

Keywords: machine retrieval; hash; TRIE; uniqueness; Java; natural language processing; intelligent information processing

1. Introduction

Research indicates that the scale of data and databases in different industrial sectors grows explosively day by day; meanwhile, an increasing number of web search engines such as Google are appearing to support information search on the Web, as the Internet is reshaping the business operation pattern with its unprecedented growth and application (D'Mello and Ananthanarayana 2010, Gong *et al*. 2010, Wang *et al*. 2010, Cao and Yang 2011, Duan *et al*. 2011, Fu *et al*. 2011, Li 2011, Wang *et al*. 2011, Xu 2011). Knowledge management plays a main role in enterprise information systems now, along with the development of enterprise information systems (Qi *et al*. 2006, Xu *et al*. 2006). Information retrieval (IR) with natural language processing (NLP) is particularly important. A search engine uses an inverted index to respond to queries (Zobel and Moffat 2006). All of the pages are organised based on terms for keyword matching. In Google's architecture (Brin and Page 1998), lexicon searching is the key mechanism that is used. Meanwhile, it is necessary to check the uniqueness of visited URLs to avoid superfluous downloads.

*Corresponding author. Email: dillony@gmail.com

ISSN 1751-7575 print/ISSN 1751-7583 online

http://dx.doi.org/10.1080/17517575.2012.665483
http://www.tandfonline.com

An additional task required for a Chinese, Japanese and Korean (CJK) search engine is CJK segmentation. For CJK pages, there is no delimiter between words. Therefore, segmentation is the basis of CJK information processing, while lexicon searching is the basis of segmentation.

The TRIE algorithm has been used widely in information processing, which is applicable to above-mentioned tasks. The name 'TRIE' is derived from the word retrieval. Fredkin (1960) coined this term for TRIE memory and Knuth (1997) introduced the classical algorithm on TRIE.

To match a pattern, a TRIE partitions all of the patterns into segments and stores them in a tree structure. Every child of a node has the same prefix. Matching of a pattern consists of every match from the root to a leaf. Figure 2(a) shows a Chinese dictionary in which, for example, node '大气层 (daqiceng)' and node '大气的 (daqide)' are children of node ' 气 (qi).' The matching of ' 大气电离 (daqi dianli)' will pass the four nodes: ' 大 (da),' ' 气 (qi),' ' 电 (dian)' and ' 离 (li).' In NLP, it is always necessary to seek a cluster of terms that have the same prefixes or suffixes. In this regard, a TRIE algorithm can provide what normal algorithms cannot provide.

The number in the square brackets in Figure 2(b) indicates the number of children a node has. The most active characters have considerable capacity to compose words, which can necessitate thousands of Chinese characters. The simple TRIE algorithm is not suitable for lexicon search, CJK segmentation and URL identification. The dynamic Hash TRIE (DH-TRIE) has a concise model for performing matching in an economical manner (Yang *et al*. 2008); hash tables help each node find every child node rapidly. Because the creation of a node needs a memory management block in an object-oriented programming language, the memory overhead has been a problem.

In this article, to promote the space utility of the naïve DH-TRIE, the Auto-Array is developed to change its size on demand without redundant restructuring; objects derived from a class can be converted in Auto-Arrays, whose number is based on the number of members that the class has. As the hash table is the main factor of occupying memory, the linked-list–based hash table is replaced with the Hash-Array, which performs better in terms of space. After integrated with the Auto-Array and the Hash-Array, the enhanced DH-TRIE significantly improves space utility. The rest of the article is organised as follows. Section 2 describes the related work, Section 3.1 presents the Auto-Array algorithm, Section 3.2 presents the Hash-Array algorithm, Section 3.3 proposes the enhanced DH-TRIE algorithm, and Section 4 provides empirical results. Section 5 finally concludes the article.

2. Related work

In recent years, an increasing number of researches have been conducted on TRIE. Dundas (1991) provided a modified TRIE-searching algorithm named Keyword TRIE, in which the choice of child nodes is made by visiting the chain of all of the children. TRIE is suitable for 26 characters and other notations in English, but not for large CHARSET such as Chinese (Morimoto *et al*. 1994). Sun *et al*. (2000) introduced three typical dictionary mechanisms, binary-seek-by-word, TRIE indexing tree and binary-seek-by-characters (BSBC), and considered that the BSBC is the most appropriate among them. Li *et al*. (2003) provided a new dictionary mechanism called double-character-hash-indexing (DCHI), which

improves the speed and efficiency of segmentation. Aoe *et al*. (1992) introduced the double-array to resolve the space problem found in TRIE, which has been applied to Chinese retrieval; the results showed that the space utility is excellent. However, it does not perform as well in aspects of scalability, making it difficult to add new words or to revise the dictionary (Li *et al*. 2006, Wang *et al*. 2006). Zhai *et al*. (2007) provided a dynamic TRIE tree in allusion to maintain new words. The Bloom filter is a fast hash-coding method that can be used for URL identification (Bloom 1970). It has a good recall ratio that can ensure that every term in the dictionary will be found, but it allows for errors, which decreases its precision ratio. Furthermore, the filter requires dedicated storage for the messages or terms. All of these studies mention that TRIE has high efficiency in terms of time but low efficiency in space.

Yang *et al*. (2008) proposes a solution that uses a hash table to locate a child node with Unicode in the DH-TRIE. The time required for matching a character in a term is nearly constant, regardless of the number of child nodes. Figure 1 shows the insertion of the DH-TRIE. A term list is shown in Figure 1(a). The structure of the DH-TRIE after the first term *address* is shown in Figure 1(b); the node *root* is simply a portal of the DH-TRIE. To the second term *add* (Figure 1(c)), the DH-TRIE finds the child node 1, which contains the same prefix *a*, so node 1 should be split into nodes 2 and 1. When the terms *overhead* and *cost* occur (Figure 1(d)), the DH-TRIE cannot find any match and generates two new nodes, 3 and 4. When the term *addendum* occurs (Figure 1(e)), the DH-TRIE finds the child node 2, which contains the same prefix *add*, and *addendum* matches all characters of node 2, so the left end of the word should be put into a new node 5 as a child of node 2. When the term *charge* occurs (Figure 1(f)), the DH-TRIE finds the node 4 with the prefix *c* and the remainder is not matched. Therefore, the DH-TRIE splits node 4 into nodes 6 and 4 and puts the remainder of *charge* into a new node 7 as a child of node 6.

The structure for nodes in the naïve DH-TRIE is a 5-tuple TrieNode = (*Body*, *Parent*, *Sons*, *IsTerm*, *tag*), where the *Body* is a segment of a term, whose first character is different from its siblings' node; the *Parent* ∈ TrieNode is a reference to the parent node of this node, by which the DH-TRIE can visit the siblings for semantic purpose and return the complete term by concatenation of all of the *Body*

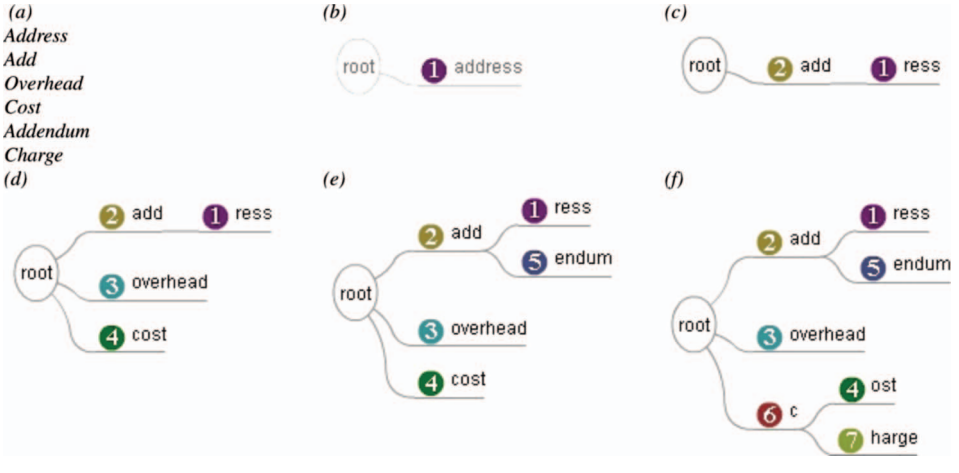

Figure 1. Example of insert terms in the DH-TRIE.

one by one up to the root, which is useful for a semantic search by relationship between nodes; the *Sons* is a mapping Sons: Character → TrieNode, which is implemented by java.util.HashMap; the *IsTerm* is a flag appended to a term; the *tag* is the data field, which is user-defined type.

Performing semantic searches in the DH-TRIE is simple. Assuming that the data field *tag* contains semantic associations (Figure 1(f)): the *tag* of node 4 contains 3 and 7; the *tag* of node 3 contains 4 and 7 and the *tag* of node 7 contains 3 and 4. Given the term *cost*, the DH-TRIE can locate node 4 and return the semantic associations *overhead* and *charge* by concatenations of all of the *Body* one by one from node 3 and node 7 to the root.

Based on object-oriented programming, the nodes in the DH-TRIE cause low efficiency in terms of space utilisation.

3. An enhanced DH-TRIE algorithm

Assuming that the allocated memory block of an object consists of two components: Block-NEW, the constant overhead for memory management for all objects, and Block-Object, the accessible bytes for the object itself; the formula for calculating allocated memory blocks can be derived from predefined Block-Objects with

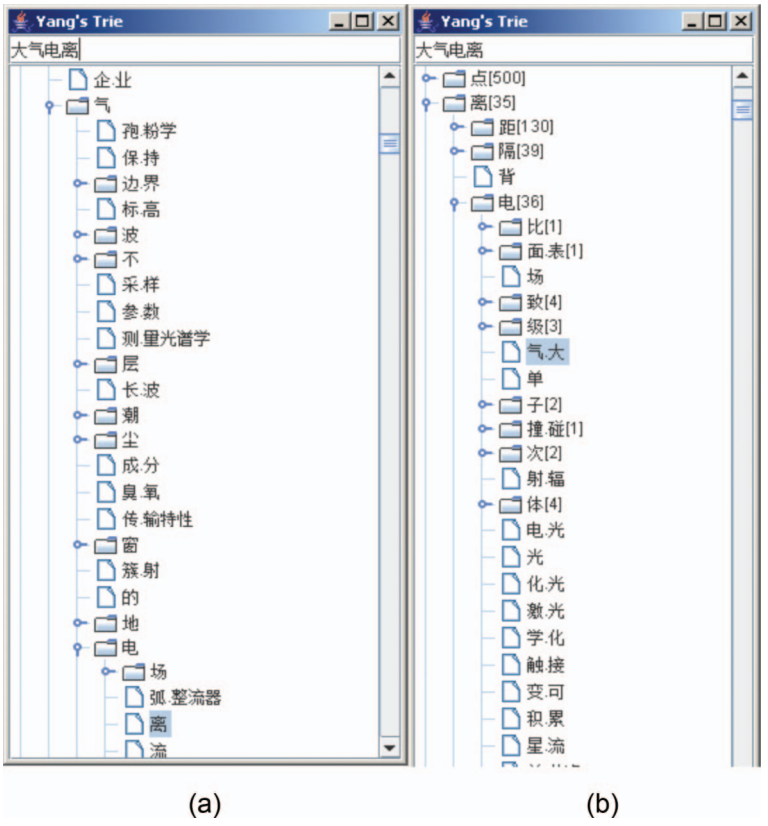

Figure 2. The examples of forward the eDH-TRIE and backward the eDH-TRIE.

different size. All the simple types had been tested for the Block-NEW in C++ and Java; the results show that the Block-NEW is 44-byte and the memory increment for an object or array is 16-byte in C++; in Java, the corresponding overheads are 12-byte and 8-byte for an array (a kind of special Java class), 8-byte and 8-byte for an object. For example, in Java, a 4-byte array occupies 20 bytes, and so does an 8-byte array, because their sizes are within an 8-byte increment; a 9-byte array occupies 28 bytes, because it takes two 8-byte increments. Given a size of an array, the formulas for calculating allocated memory block are proposed (Table 1). Therefore, the assumption that the Block-NEW is constant for all Block-Objects is proven. The nodes of the DH-TRIE (Figure 1(f)) can be represented in the arrays (Table 2). Because each node has a Block-NEW, the more rows there are, the more is the cost of memory.

Instead of generating many Block-NEWs, a few arrays are assigned based on the members, saving the overhead of memory (Table 3). An integer array called *iaParent* stores the parent ID of each node; a Boolean array called *baIsTerm* stores the information of whether this node is a term; an integer array called *iaPtrBody* stores the address of the body in the *caBody* and an integer array called *iaLenBody* stores the length of each body. A buffer called *caBody* stores all the characters in each node (Table 4).

The member *Sons* contains the children of a node, whose number varies from 0 to $0 \times$ FFFF in Unicode (Table 3). Instead of assigning a fixed space for the member *Sons* for each node, a hash table provides a flexible and high-speed method to find children. For example, the *Sons* [0] will return a reference in Java to a hash table that contains three integers, 2, 3 and 6, which help to find the children (Table 3).

In Table 3, the *NodeID* is in fact the subscript of each array to obtain data, which is not necessarily assigned to an array. For example, the member *body* of node 1

Table 1. The formulas for memory allocation.

Allocation of Java object	Allocation of Java array	Allocation of C++ object
$\left(\left\lfloor\frac{\text{MemObj}-1}{8}\right\rfloor+2\right)\times 8$	$\left(\left\lfloor\frac{\text{MemObj}-5}{8}\right\rfloor+3\right)\times 8$	$\left(\left\lfloor\frac{\text{MemObj}-5}{16}\right\rfloor+4\right)\times 16$

The MemObj in the three formulas mean Block-Object, the accessible bytes of a structure; in fact, there are a leading-area of 12, 8, 44 bytes for the overhead of management of Java object, Java array and C++ object, respectively.

Table 2. List of members of each node in the naïve DH-TRIE.

NodeID[a]	Body[b]	Parent[c]	Sons[d]	IsTerm[e]
0		Null	2,3,6	False
1	ress	2	Null	True
2	add	Root	1,5	True
3	overhead	Root	Null	True
4	ost	6	Null	True
5	endum	2	Null	True
6	c	Root	4,7	False
7	harge	6	Null	True

[a]The NodeID is the ID for an object in the list.
[b,c,d,e]The 'Body', 'Parent', 'Sons', 'IsTerm' are the members in the class of the naïve DH-TRIE.

Table 3. Arrays converted from the nodes in the naïve DH-TRIE.

NodeID[a]	iaPtrBody[b]	iaLenBody[c]	iaParent[d]	Sons[e]	baIsTerm[f]
0	−1	−1	−1	2, 3, 6	False
1	0	4	2	Null	True
2	4	3	0	1, 5	True
3	7	8	0	Null	True
4	15	3	6	Null	True
5	18	5	2	Null	True
6	23	1	0	4, 7	False
7	24	5	6	Null	True

[a]The NodeID means the subscripts of elements in the arrays.
[b,c,d,e,f]The 'iaPtrBody', 'iaLenBody', 'iaParent', 'Sons', 'baIsTerm' are arrays converted from Table 2'.

Table 4. Character buffer caBody with 32-character size (16 characters in each row).

address	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0 × 00	r	e	s	s	a	d	d	o	v	e	r	h	e	A	d	o
0 × 10	s	t	e	n	d	u	m	c	h	a	r	g	e			

(Table 2), which contains *ress*, is replaced with the combination of a pointer of 0 × 00 (Table 4) and a length of 4 in the buffer *caBody*. The member *parent* of node 1 (Table 2) is replaced with the subscript of node 2. Therefore, the parent of node 1 can be obtained by *iaParent*[1] = 2.

The creation of an object can occur on demand, which is different from the predefined length of an array. It is necessary to handle arrays with increasing or decreasing elements, rather than using fixed-size arrays. There are some packages such as java.util.ArrayList in Java language that can change the size of the array when the limit is reached. However, the existing algorithm packages have restructuring operations to adapt the growing data that are beyond the initial allocated memory block. The method in these algorithms is to rebuild an old array into a new array, hence is not good at time performance. The CArray in MFC,[1] such as CUIntArray, CByteArray, CDWordArray, CWordArray, CPtrArray and the TList[2] in Borland C++ Builder do the same thing.

3.1. Auto-Array

To maintain an arbitrary size array without redundant restructuring, a structure called Auto-Array is proposed. The Auto-Array is an 8-tuple (*tag*, *tBlock*, *iTotalBlocks*, *iSizeofBlock*, *iBlockCurrent*, *iSubCurrent*, *get*, *set*), where the *tag* is the user-defined type; the *tBlock* is a 2D array of the *tag* (Figure 3(b)), i.e. multiple 1D arrays, which may not be in a sequential memory blocks; the *iTotalBlocks* is the size of the first dimension array, which is the *m* (Figure 3(b)), i.e. the number of the 1D arrays; the *iSizeofBlock* the size of the second dimension arrays, also is the *n* in Figure 3(b); the *iBlockCurrent* is a integer which indicates the current 1D array in the *tBlock*; the *iSubCurrent* is a integer which indicates the current *tag* in a 1D array; the *get* is a function: integer → *tag*, which calculates the *quotient* and *remainder* of an integer address, say *pos*, by the divisor *iSizeofBlock* and returns the

tag element *tBlock*[*quotient*][*remainder*]; the *set* is a procedure, which sets an element at an defined address to an defined value by the same address calculating rule as the *get*.

The *iTotalBlocks* and the *iSizeofBlock* can be set so large that the Auto-Array can overflow the physical memory. The Auto-Array initially allocates only one n-size array. When the size increases or decreases, the Auto-Array can allocate or delete the corresponding 1D array. Therefore, the Auto-Array can change its size on demand without redundant restructuring, in a step of the *iSizeofBlock*.

The *quotient* and *remainder* are calculated by shift operation for speed, accordingly the *iSizeofBlock* is always a power of 2. Given *iSizeofBlock* = 512, for example, *set*(513, *K*) will modify the element *tBlock*[1][0], the first element of the second 1D array, to the value of *K*; *get*(513) will return the element *tBlock*[1][0].

3.2. Hash-Array

Hash is a well-known algorithm usually implemented by linked lists. The java.util.HashMap uses an array as barrels and uses linked lists to store the key-value pairs (Lea *et al*. 2006). Java.util.HashMap maintains barrels whose number is a power of 2, such as 8, 512 and 4096 (Figure 4). If the key is 9 and the value is *V9*, for example, then a new pair should be created and put into the barrel 1 because 9 mod 8 = 1, as is done with the pair of 1 and *V1*. If the capacity grows, java.util.HashMap will rebuild the barrels to fit a capacity factor that is default 0.75. The number of pairs in a barrel is not limited to linked-lists. The problem is with the occurrence of many Block-NEWs for the pairs.

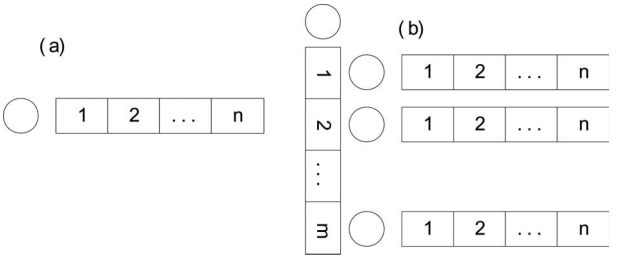

Figure 3. Illustration of the structure of Auto-Array. The circles suggest the overheads of Block-NEW; the squares suggest the overheads of objects.

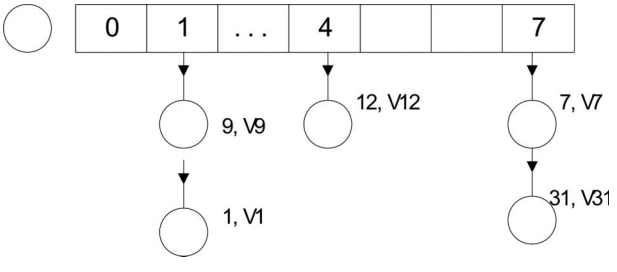

Figure 4. Illustration of Hash structure in java.util.HashMap.

A new form of hash algorithm, called Hash-Array, is proposed based on Auto-Array (Figure 5). The Hash-Array is a 6-tuple (*iCount*, *iThreshold*, *caKey*, *iaValue*, *iTotalBlocks*, *iSizeofBlock*), where the *iCount* is the count of pairs in the Hash-Array, i.e. the size of it; the *iThreshold* is the threshold that limit the *iCount* will not over 75% of the real capacity to avoid hash collisions; the *caKey* and *iaValue* are two Auto-Arrays that store keys and values, respectively; the *iTotalBlocks* and *iSizeofBlock* are the parameters of both *caKey* and *iaValue*, that keep the two Auto-Arrays in the same size. The *iTotalBlocks* can be regarded as the length of a linked list in a normal hash, the *iSizeofBlock* as the number of barrels.

Searching a key in a long linked-list is not optimal, so do the Hash-Array. The *iTotalBlocks* is fixed to a defined size, say 5; the *iSizeofBlock* is a power of 2 that can be enlarged or shrunk on demand. The key-value pairs in Figure 5 will be put into the arrays at corresponding positions. The 1D array will be created on demand and its number will not go beyond the limit 5. If a sixth pair in a barrel occurs, then the Hash-Array will double the *iSizeofBlock* and rebuild the arrays to ensure no more than five pairs in a barrel.

In fact, the nodes that have only a few child nodes are majority. Considering cache performance, a cache block is usually 64 bytes (Przybylski 1990, Lam *et al.* 1991). If a small number of children are put in a normal array, instead of Auto-Array, sequential searching in a cache block will be best for performance. Therefore, the *caKey* and *iaValue* will be normal arrays when the *iCount* is less than a suitable size, called *iTransformer*, and will be Auto-Arrays when the *iCount* is greater than the size. For integers in Java, 13 is the best choice of the *iTransformer*, because 12-byte memory management plus 13 4-byte integers equals 64 bytes by our formula (Table 1).

The final form of Hash-Array is a 7-tuple (*iCount*, *iThreshold*, *caKey*, *iaValue*, *iTotalBlocks*, *iSizeofBlock*, *iTransformer*). The *iTransformer* is empirically set to 13, based on the size of cache block. If the size of a Hash-Array is no more than 13, two 13-size 1D arrays will be set for sequential searching in a cache block; if the size is over 13, the Hash-Array sets two Auto-Arrays to perform hash searching.

Table 5 shows the algorithm for putting a new pair into the array. The *caK* and *iaV* are the two 1D arrays for small hash tables. When the size is more than 13, the two 1D arrays will be transferred to 2D Hash-Array and be deleted.

3.3. *Integrated with Auto-Array and Hash-Array*

The enhanced DH-TRIE is proposed as a 7-tuple eDH-TRIE = (*caBody*, *iaPtrBody*, *iaLenBody*, *iaParent*, *hSons*, *baIsTerm*, *oaTag*); where the *caBody* is an Auto-Array

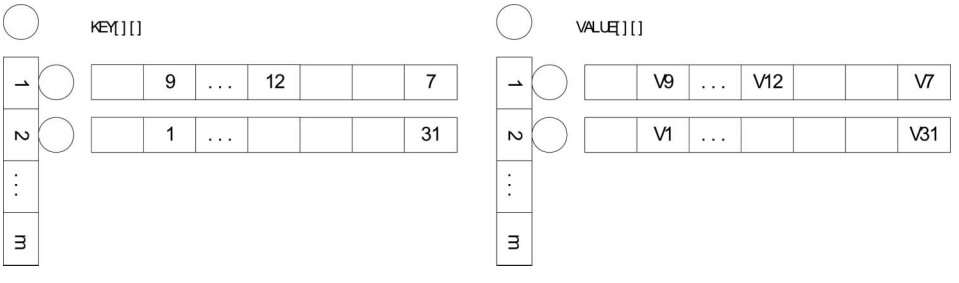

Figure 5. Illustration of the Hash-Array structure.

Table 5. Algorithm for putting key and value pairs into a Hash-Array.

FUNCTION put(cK, iV)
1. **IF** Auto-Array exist **THEN** put (cK, iV) into Auto-Array;
2. **ELSE**
   a. **FOR** i = 0 **THRU** iBoundHash **BY** 1 **DO**
      1) **IF** caK[i] = cK **THEN** iaV[i] ← iV, **RETURN**;
      2) **IF** caK[i] = 0 **THEN** caK[i] ← cK, iaV[i] ← iV, count ← count + 1, **RETURN**;
   b. transfer the 1D array to Auto-Array, put (cK, iV) into Auto-Array;

of term segments, whose first character is different from its siblings; the *iaPtrBody* is an Auto-Array of pointers which indicate the position of a term segment in the *caBody*; the *iaLenBody* is an Auto-Array of lengths which indicate the length of a term segment in the *caBody*; the *iaParent* is an Auto-Array of integers which indicate the subscripts of parent nodes; the *hSons* is an Auto-Array of Hash-Arrays, which are mappings, *hSon*: Character → integer; the *baIsTerm* is an Auto-Array of Boolean flags; the *oaTag* is an Auto-Array of user-defined type.

In Table 9, the *NORM 2D* is the NORM according to the DH-TRIE with a Hash-Array exclusively and the *NORM 1D* is the NORM according to the DH-TRIE with 1D 32-size array for small hash tables, which avoid the overhead of Hash-Arrays. The *URLS 1D* is the URL identification experiment by the DH-TRIE with 1D 16-size array for a small hash table, which eliminates most of the Hash-Arrays; meanwhile, the performance is the same with *URLS 2D*. The 'overhead xxx' are calculated by the algorithm.

The 'memory calculated' is based on 8-byte memory increments, which is less than the 'Mem_Observed,' because of the cost for the Java code itself. In any case, the space utility has been improved significantly compared to the data in Table 10. The '1D array' means the number of small hash tables; the DH-TRIE does sequential locating in this limit to avoid overhead of memory. It is a tradeoff between saving space and saving time.

3.4. Result of CJK segmentation

Because the eDH-TRIE ignores unmatched part of a term, it will suit for CJK segmentation. The forward maximal matching is one of the worst algorithms in CJK segmentation, but it still works well with the eDH-TRIE, for example:

Input 1: 第三条公司是企业法人，有独立的法人财产，享有法人财产权。公司以其全部财产对公司的债务承担责任。(disantiao gongsi shi qiye faren you dulide faren caichan xiangyou faren caichanquan gongsi yiqi quanbu caichan dui gongsi de zhaiwu chengdan zeren)

Result 1 (forward): [第三] [条] [ ] [公司] [是] [企业法人] [,] [有] [独立的] [法人财产] [,] [享有] [法人] [财产权] [。] [公司] [以其] [全部] [财产] [对] [公司的] [债务] [承担责任] [。] [ ]

Input 2: 公路局正在治理解放大道路面积水问题 (gongluju zhengzai zhili jiefang dadao lumian jishui wenti)

Result 2 (forward): [公路] [局] [正在] [治理] [解放] [大道] [路面积水] [问题]

Result 3 (backward): [公] [路局] [正在] [治理] [解放] [大道] [路面积水] [问题]

The advantage of large lexicon is that the uncommon terms as "路面积水 (lumian jishui)" can be recognized. Note that the combinations [放大] [道路] [面积] are possible phrases.

The advantage of large lexicon is that the uncommon terms as '成吉思汗 (Chengjisihan)' and '窝阔台 (wokuotai)' can be recognised. If we build an inverted the eDH-TRIE that contains all the inverted terms as Figure 2(b), we can use the backward maximal matching to get better accuracy.

4. Experiments and results

To test the performance of the eDH-TRIE, the following experiments were designed to compare it with the naïve DH-TRIE, java.util.HashSet and java.util.TreeSet. Environment: OS: windows XP sp2; Programming language: JDK1.5, VC6.0; CPU: Xeon 2.4 GHz; Memory: 1 GB.

The experiment for the Auto-Array was to test the performance in changing without restructuring, compared with java.util.ArrayList; the Auto-Array was superior in all aspects (Table 6). When the size of idle memory was 696330 KB, the Auto-Array used almost the total memory; meanwhile, java.util.ArrayList could not allocate the margin between 623072 and 696330 because the size of its old block was larger than the margin. Java.util.ArrayList took more time than Auto-Array because it repeated the restructuring operation. The most important feature was that Auto-Array can allocate about 6.5 times more elements than java.util.ArrayList, because it cancelled most of the *Block-NEW*s. Therefore, the average cost for each integer was 4.164 vs. 24.764.

The experiment for the Hash-Array was to test the performance in terms of time and space, compared with java.util.HashMap (Table 7). To test the adding and removing performance, 0 × 10,000 Character-Integer pairs had been put into and

Table 6. Comparison of Auto-Array with 0 × 1000 * 0 × 10,000 size and java.util.ArrayList in generating dynamic integer arrays.

Performs	Auto-Array	java.util.ArrayList
Number of integer[a]	168,132,608	25,763,840
Idle memory (KB)[b]	696,330	696,330
Mem_Observed (KB)[c]	680,764	623,072
Time (ms)	9683	20,383
bytes/integer	4.164	24.764

[a]The 'number of integer' means the number of generated integers.
[b,c]The 'Idle memory' and 'Mem_Observed' is the data observed from task management in the windows XP.

Table 7. Comparison of Hash-Array and java.util.HashMap for inserting of 0 × 10,000 Character-Integer pairs.

Performs	Hash-Array	java.util.HassMap
Mem_Observed (KB)[a]	1088	5052
Time adding (ms)[b]	31	125
Time removing (ms)[c]	31	15

[a]The abbreviations as in Table 6.
[b]The abbreviations as in Table 9.
[c]The 'time removing' means the time cost for removing all pairs.

removed from the Hash-Array and java.util.HashMap. The former used less memory and less time in adding.

The experiments for eDH-TRIE were based on the two lexicon searching tasks, NORM and URLS. Because there exists no reasonable norm for evaluating all the lexicon algorithms, an algorithm NORM is presented in Table 8, which generates all strings whose length is the *LEN* (Yang *et al*. 2008). If the *LEN* equals 4, it can generate $26^4 = 456{,}976$ strings; this number is close to a medium lexicon. URLS, the URL identification experiment, has 64,994 URLs in one website to store and to search (Yang *et al*. 2008).

The results show that the eDH-TRIE will perform better in terms of both time and space if the small hash tables are converted into 1D arrays (Table 9). Table 10 shows that the naïve DH-TRIE has the similar performances of the

Table 8. NORM: algorithm for testing the performance of lexicon algorithms (Yang *et al*. 2008).

```
FUNCTION NORM(int len)
1. char Str[] ← new char[len];
2. addchars(Str,0,Str.length);
FUNCTION addchars(char Str[], int layer, int limit)
3. IF layer beyond limit THEN add the string into dictionary, RETURN;
4. FOR each ch ∈ {a-z} DO Str[layer] ← ch; addchars(Str,layer + 1,limit);
```

Table 9. Performances of eDH-TRIE in time and space.

Performances	NORM 2D[a]	NORM 1D[b]	URLS 2D[c]	URLS 1D[d]
Time adding (ms)[e]	671	609	609	516
Time finding (ms)[f]	266	312	359	343
Total nodes[g]	475,255	475,255	85,646	85,646
Memory calculated (KB)[h]	17,318	14,462	6952	6741
Mem_Observed (KB)[i]	26,272	22,996	14,664	14,424
1D array[j]	0	18,279	15,839	20,641
2 barrels[k]	0	0	4534	0
4 barrels[l]	0	0	261	0
8 barrels[m]	18,279	0	18	11
16 barrels[n]	0	0	2	2
32 barrels[o]	0	0	0	0

[a]The 'NORM 2D' is the NormTest by the eDH-TRIE with the Hash-Array exclusively.
[b]The 'NORM 1D' is the NormTest by the eDH-TRIE with the 1D 32-size array for the small hash tables, which avoid the overhead of the Hash-Arrays.
[c]The 'URLS 1D' is the URL identification experiment by the eDH-TRIE with the 1D 16-size array for the small hash table, which eliminate most of the Hash-Arrays, meanwhile the Performance is the same with 'URL with 2D'.
[d]The 'URLS 2D' is the URL identification experiment by the eDH-TRIE with the Hash-Array exclusively.
[e]The 'Time adding' is the time for adding all the terms into the algorithms.
[f]The 'Time finding' is the time for finding all the terms from the algorithms.
[g]The 'Total nodes' is the number of all the nodes in the eDH-TRIE.
[h]The 'memory calculated' is total overhead of the eDH-TRIE based on the algorithm in Table 1.
[i]The abbreviations as in Table 6.
[j]The '1D array' is the number of the small hash tables with 1D arrays in the eDH-TRIE.
[k,l,m,n,o]The 'xxx barrels' are the number of the barrels of the Hash-Array with size of 2, 4, 8, 16, 32, respectively.

Table 10. Performances of URLS and NORM experiments.

	URLS[a]				NORM[b]			
Performance	DH-TRIE[c]	eDH-TRIE[d]	HashSet[e]	TreeSet[f]	DH-TRIE	eDH-TRIE	HashSet	TreeSet
Mem_Observed (KB) [g]	19,200	14,424	25,020	24,820	59,516	22,996	50,020	47 328
Time adding (ms)[h]	641	516	500	610	1766	609	1,906	1 656
Time finding (ms)[i]	297	343	250	265	235	312	406	453

[a]The 'URLS' is the URL identification experiment.
[b]The 'NORM' is the experiment NORM as in Table 8.
[c]The 'DH-TRIE' is the naïve DH-TRIE.
[d]The 'eDH-TRIE' is the enhanced DH-TRIE.
[e,f]The 'HashSet' and the 'TreeSet' are the packages in Java JDK.
[g,h,i]The abbreviations as in Table 6.

packages java.util.HashSet and java.util.TreeSet. The eDH-TRIE improves the space performance more than the others while keeps the average time performance.

5. Conclusion

IR is essential to enterprise systems along with the scale of data and databases in different industrial sectors. Efficiency of retrieval system is the motivation of this article. To resolve the problem of memory overhead in the naïve DH-TRIE, we propose the Auto-Array and Hash-Array algorithms. The Auto-Array can maintain a dynamic array without redundant restructuring; it performs better in the comparison experiment. The Hash-Array based on the Auto-Array is for reducing the memory management overheads, the result shows it takes less memory. A new algorithm called enhanced dynamic hash TRIE (eDH-TRIE) is presented, which stores objects by the Auto-Arrays and searches child nodes via the Hash-Array. We have demonstrated that the algorithm performs extremely well with flexible extensibility for the lexicon searching and URL identification. In addition to the DH-TRIE, the Auto-Array and the Hash-Array can be used in a multitude of situations. The CJK segmentation with eDH-TRIE is a concise and efficient approach for NLP. In the future, we will focus on a CJK semantic indexing system in enterprise systems.

Acknowledgements

This work was partially supported by the Chinese Academy of Sciences under Grant No. 20040402 (Distinguished Overseas Scholar Grant), Changjiang Scholar Program of the Ministry of Education of China, National Natural Science Foundation of China under Grant Nos. 61035003, 60903141, 61072085, 60933004, 71132008 National Basic Research Priorities Programme under Grant No. 2007CB311004 and US National Science Foundation under Grant 1044845.

Notes

1. 'CArray Classes,' http://msdn.microsoft.com/en-us/library/4h2f09ct (VS.80).aspx ed: Microsoft Corporation, 2008.
2. 'Borland C++ Builder Help,' C++ Builder Version 6.0 (Build 10.157) ed: Borland Software Corporation, 2006.

References

Aoe, J.I., Morimoto, K., and Sato, T., 1992. An efficient implementation of trie structures. *Software – Practice and Experience*, 22 (9), 695–721.

Bloom, B.H., 1970. Space/time trade/offs in hash coding with allowable errors. *Communications of the ACM*, 13 (7), 422–426.

Brin, S. and Page, L., 1998. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30 (1–7), 107–117.

Cao, X. and Yang, F., 2011. Measuring the performance of internet companies using a two-stage data envelopment analysis model. *Enterprise Information Systems*, 5 (2), 207–217.

D'Mello, D.A. and Ananthanarayana, V.S., 2010. Dynamic selection mechanism for quality of service aware web services. *Enterprise Information Systems*, 4 (1), 23–60.

Duan, L., Street, W.N., and Xu, E., 2011. Healthcare information systems: data mining methods in the creation of a clinical recommender system. *Enterprise Information Systems*, 5 (2), 169–181.

Dundas, J.A., 1991. Implementing dynamic minimal-prefix tries. *Software-Practice & Experience*, 21 (10), 1027–1040.
Fredkin, E., 1960. Trie memory. *Communications of the ACM*, 3 (9), 490–499.
Fu, C., *et al*., 2011. Study on the contract characteristics of internet architecture. *Enterprise Information Systems*, 5 (4), 495–513.
Gong, Z., Muyeba, M., and Guo, J., 2010. Business information query expansion through semantic network. *Enterprise Information Systems*, 4 (1), 1–22.
Knuth, D., 1997. Digital searching. *In*: *The art of computer programming*, Vol. Sorting and Searching, Beijing: Tsinghua University Press & Addison-Wesley, 492–512.
Lam, M.S., Rothberg, E.E., and Wolf, M.E., 1991. The cache performance and optimizations of blocked algorithms. *Paper presented at the 4th international conference on architectural support for programming languages and operating systems*, 8–11 April. Santa Clara, CA: Assoc Computing Machinery.
Lea, D., *et al*., 2006. 'HashMap,' JDK 1.5 ed: Sun Microsystems, Inc.
Li, L., 2011. Introduction: advances in e-business engineering. *Information Technology and Management*, 12 (2), 49–50.
Li, J.b., Zhou, Q., and Chen, Z.s., 2006. A study on fast algorithm for Chinese dictionary lookup. *Journal of Chinese Information Processing*, 20 (5), 31–39.
Li, Q.h., Chen, Y.j., and Sun, J.g., 2003. A new dictionary mechanism for Chinese word segmentation. *Journal of Chinese Information Processing*, 17 (4), 13–18.
Morimoto, K., Iriguchi, H., and Aoe, J.I., 1994. A method of compressing trie structures. *Software, Practice & Experience*, 24 (3), 265–288.
Przybylski, S.A., 1990. *Cache and memory hierarchy design: A performance directed approach*. San Mateo: Morgan Kaufmann.
Qi, J., *et al*., 2006. Knowledge management in OSS – an enterprise information system for the telecommunications industry. [10.1002/sres.752]. *Systems Research and Behavioral Science*, 23 (2), 177–190.
Sun, M.s., Zuo, Z.p., and Huang, C.n., 2000. An experimental study on dictionary mechanism for Chinese word segmentation. *Journal of Chinese Information Processing*, 14 (1), 1–6.
Wang, L., Zeng, J., and Xu, L., 2011. A decision support system for substage-zoning filling design of rock-fill dams based on particle swarm optimization. *Information Technology and Management*, 12 (2), 111–119.
Wang, S.l., Zhang, H.p., and Wang, B., 2006. Research of optimization on double-array TRIE and its application. *Journal of Chinese Information Processing*, 20 (5), 24–30.
Wang, K., *et al*., 2010. A service-based framework for pharmacogenomics data integration. *Enterprise Information Systems*, 4 (3), 225–245.
Xu, L., 2011. Enterprise systems: state of the art and future trends. *IEEE Transactions on Industrial Informatics*, 7 (4), 630–640.
Xu, L., *et al*., 2006. Integrating knowledge management and ERP in enterprise information systems. [10.1002/sres.750]. *Systems Research and Behavioral Science*, 23 (2), 147–156.
Yang, L., *et al*., 2008. Research and analysis of dynamic hash TRIE algorithm. *Journal of Guangxi Normal University, China, Natural Science Edition*, 26 (1), 134–138.
Zhai, W.b., Zhou, Z.l., and Jiang, Z.m., 2007. Design dictionary of Chinese word segmentation. *Computer Engineering and Applications*, 43 (1), 1–2, 26.
Zobel, J. and Moffat, A., 2006. Inverted files for text search engines. *ACM Computing Surveys*, 38 (2), 1–56.