PAT-Trees with the Deletion Function as the Learning Device for Linguistic Patterns

Keh-Jiann Chen, Wen Tsuei, and Lee-Feng Chien
CKIP, Institute of Information Science,
Academia Sinica, Nankang, Taipei 115, Taiwan

Abstract

In this study, a learning device based on the PAT-tree data structures was developed. The original PAT-trees were enhanced with the deletion function to emulate human learning competence. The learning process worked as follows. The linguistic patterns from the text corpus are inserted into the PAT-tree one by one. Since the memory was limited, hopefully, the important and new patterns would be retained in the PAT-tree and the old and unimportant patterns would be released from the tree automatically. The proposed PAT-trees with the deletion function have the following advantages. 1) They are easy to construct and maintain. 2) Any prefix sub-string and its frequency count through PAT-tree can be searched very quickly. 3) The space requirement for a PAT-tree is linear with respect to the size of the input text. 4) The insertion of a new element can be carried out at any time without being blocked by the memory constraints because the free space is released through the deletion of unimportant elements.

Experiments on learning high frequency bi-grams were carried out under different memory size constraints. High recall rates were achieved. The results show that the proposed PAT-trees can be used as on-line learning devices.

1. Introduction

Human beings remember useful and important information and gradually forget old and unimportant information in order to accommodate new information. Under the constraint of memory capacity, it is important to have a learning mechanism that utilizes memory to store and to retrieve information efficiently and flexibly without loss of important information. We don't know how human memory functions exactly, but the issue of creating computers with similar competence is one of the most important problems being studied. We are especially interested in computer learning of linguistic patterns without the problem of running out of memory.

To implement such a learning device, a data structure, equipped with the following functions, is needed: a) accept and store the on-line input of character/word patterns, b) efficiently access and retrieve stored patterns, c) accept unlimited amounts of data and at the same time retain the most important as well as the most recent input patterns. To meet the above needs, the PAT-tree data structure was originally considered a possible candidate to start with. The original design of the PAT-tree can be traced back to 1968. Morrison [Morrison, 68] proposed a data structure called the "Practical Algorithm to Retrieve Information Coded in Alphanumeric"(PATRICIA). It is a variation of the binary search tree with binary representation of keys. In 1987, Gonnet [Gonnet, 87] introduced semi-infinite strings and stored them into PATRICIA trees. A PATRICIA tree constructed over all the possible semi-infinite strings of a text is then called a PAT-tree. Many kinds of searching functions can be easily performed on a PAT-tree, such as prefix searching, range searching, longest repetition searching and so on. A modification of the PAT-tree was done to fit the needs of Chinese processing in 1996 by Hung [Hung, 96], in which the finite strings were used instead of semi-infinite strings. Since finite

strings are not unique in a text as semi-infinite strings are, frequency counts are stored in tree nodes. In addition to its searching functions, the frequencies of any prefix sub-strings can be accessed very easily in the modified PAT-tree. Hence, statistical evaluations between sub-strings, such as probabilities, conditional probabilities, and mutual information, can be computed.

It is easy to insert new elements into PAT-trees, but memory constrains have made them unable to accept unlimited amounts of information, hence limiting their potential use as learning devices. In reality, only important or representative data should be retained. Old and unimportant data can be replaced by new data. Thus, aside from the original PAT-tree, the deletion mechanism was implemented, which allowed memory to be released for the purpose of storing the most recent inputs when the original memory was exhausted. With this mechanism, the PAT-tree is now enhanced and has the ability to accept unlimited amounts of information. Once evaluation functions for data importance are obtained, the PAT-tree will have the potential to be an on-line learning device. We review the original PAT-tree and its properties in section 2. In section 3,we describe the PAT-tree with deletion in detail. In section 4, we give the results obtained after different deletion criteria were tested to see how it performed on learning word bi-gram collocations under different sizes of memory. Some other possible applications and a simple conclusion are given in the last section.

2. The Original PAT-tree

In this section, we review the original version of the PAT-tree and provide enough background information for the following discussion.

2.1 Definition of Pat-tree

2.1.1 PATRICIA

Before defining the PAT-tree, we first show how PATRICIA works.

PATRICIA is a special kind of trie[Fredkin 60]. In a trie, there are two different kinds of nodes—branch decision nodes and element nodes. Branch decision nodes are the search decision-makers, and the element nodes contain real data. To process strings, if branch decisions are made on each bit, a complete binary tree is formed where the depth is equal to the number of bits of the longest strings. For example, suppose there are 6 strings in the data set, and that each is 4 bits long. Then, the complete binary search tree is that shown in Fig. 2.1.

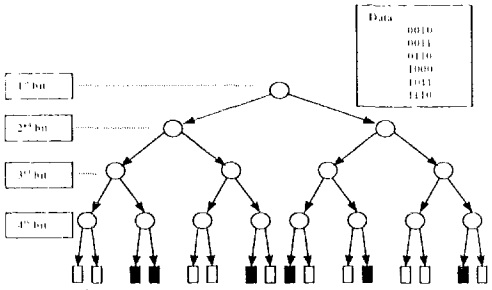

Fig 2.1 The complete binary tree of the 6 data

Apparently, it is very wasteful. Many element nodes and branch nodes are null. If those nodes are removed, then a tree called a "compressed digital search trie" [Flajolet 86], as shown in Fig. 2.2, is formed. It is more efficient, but an additional field to denote the comparing bit for branching decision should be included in each decision node. In addition, the searched results may not exactly match the input keys, since only some of the bits are compared during the search process. Therefore, a matching between the searched results and their search keys is required.

Morrison [Morrison, 68] improved the trie structure further. Instead of classifying nodes into branch nodes and element nodes, Morrison combined the above two kinds of nodes into a uniform representation, called an augmented branch node. The structure of an augmented branch node is the same as that of a decision node of the trie except that an additional field for storing elements is included. Whenever an element should be inserted, it is inserted "up" to a branch node instead of creating a new element node as a leaf node. For example, the compressed digital search trie shown in Fig 2.2 has the equivalent PATRICIA like Fig 2.3. It is noticed that each element is stored in an upper node or in itself. How the data elements are inserted will be discussed in the next section. Another difference here is the additional root node. This is because in a binary tree, the number of leaf nodes is always greater than that of internal nodes by one. Whether a leaf node is reached is determined by the upward links.

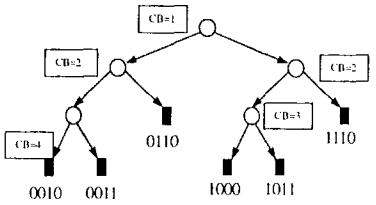

Fig. 2.2 Compressed digital search trie.

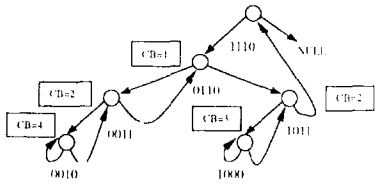

Fig 2.3 PATRICIA

2.1.2 PAT-tree

Gonnet [Gonnet, 87] extended PATRICIA to handle semi-infinite strings. The data structure is called a PAT-tree. It is exactly like PATRICIA except that storage for the finite strings is replaced by the starting position of the semi-infinite strings in the text.

Suppose there is a text T with n basic units. $T = u_1 u_2 ... u_n$. Consider the prefix sub-strings of T's which start from certain positions and go on as necessary to the right, such as $u_1 u_2 ... u_n ....$, $u_2 u_3 ... u_n ....$, $u_3 u_4 ... u_n ....$ and so on. Since each of these strings has got an end to the left but none to the right, they are so-called semi-infinite strings. Note here that whenever a semi-infinite string extends beyond the end of the text, null characters are appended. These null characters are different from any basic units in the text. Then, all the semi-infinite strings starting from different positions are different. Owing to the additional field for comparing bits in each decision node of PATRICIA, PATRICIA can handle branch decisions for the semi-infinite strings (since after all, there is only a finite number of sensible decisions to separate all the elements of semi-infinite strings in each input set). A PAT-tree is constructed by storing all the starting positions of semi-infinite strings in a text using PATRICIA.

There are many useful functions which can easily be implemented on PAT-trees, such as prefix searching, range searching, longest repetition searching and so on.

```
Insert(to-end substring Sub, PAT tree rooted at R)
{
// Search Sub in the PAT tree
      p ← R;
      n ← Left ( p );
      while ( CompareBit ( n ) > CompareBit ( p ) ) {
            p ← n;
            if the same bit as CompareBit ( p ) at Sub is 0
                  n ← Left ( p );
            else
                  n ← Right ( p );
      }
      if ( Data ( n ) = Sub ) {
      // Sub is already in the PAT tree, just
      // increase the count. No need to insert.
            Occurrence ( n ) ← Occurrence ( n ) + 1;
            return;
      }
// Find the appropriate position to insert Sub //
into the PAT tree (Sub will be inserted
// between p and n)
      b ← the first bit where Data ( n ) and Sub differ;
      p ← R;
      n ← Left ( p );
      while ( (CompareBit ( n ) > CompareBit ( p ) ) and
              (CompareBit ( p ) < b) ) {
            p ← n;
            if the same bit as CompareBit ( p ) at Sub is 0
                  n ← Left ( p );
            else
                  n ← Right ( p );
      }
//Insert Sub into the PAT tree, between p and n
      // Initiate a new node
      NN ← new node;
      CompareBit ( NN ) ← b;
      Data ( NN ) ← Sub;
      Occurrence ( NN ) ← 1;
      // Insert the new node
      If the bth bit of Sub is 0 {
            Left ( NN ) ← NN;
            Right ( NN ) ← n;
      }
      else {
            Left ( NN ) ← n;
            Right ( NN ) ← NN;
      }
      if n is the Left of p
            Left ( p ) ← NN;
      else
            Right ( p ) ← NN;
}
```

Algorithm 2.1 PAT tree Insertion

Hung [Hung, 96] took advantage of prefix searching in Chinese processing and revised the PAT-tree. All the different basic unit positions were exhaustively visited as in a PAT-tree, but the strings did not go right to the end of the text. They only stopped at the ends of the sentences. We call these finite strings "to-end sub-strings". In this

way, the saved strings will not necessarily be unique. Thus, the frequency counts of the strings must be added. A field denoting the frequency of a prefix was also added to the tree node. With these changes, the PAT-tree is more than a tool for searching prefixes; it also provides their frequencies.

The data structure of a complete node of a PAT-tree is as follows.

Node: a record of

Decision bit: an integer to denote the decision bit.
Frequency: the frequency count of the prefix sub-string.
Data element: a data string or a pointer of a semi-infinite string .
Data count: the frequency count of the data string.
Left: the left pointer points downward to the left sub-tree or points upward to a data node.
Right: the right pointer points downward to the right sub-tree or points upward to a data node.

End of the record.

The construction process for a PAT-tree is nothing more than a consecutive insertion process for input strings. The detailed insertion procedure is given in Algorithm 2.1 and the searching procedure in Algorithm 2.2.

```
SearchforFrequencyof ( Pattern )
{
    p ← R /*the root of PAT-tree*/;
    n ← Left ( p );
    while ( ( CompareBit ( n ) > CompareBit ( p ) ) and
            ( CompareBit ( n ) ≤ total bits of Pattern ) )
    {
        p ← n;
        if the "CompareBit ( p )"th bit of Pattern is 0
            n ← Left ( p );
        else
            n ← Right ( p );
    }
    if ( Data ( n ) ≠ Pattern )
        return 0;
    if ( CompareBit ( n ) > total bits of Pattern )
        return TerminalCounts ( n );
    else
        return Occurrence ( n );
}
```

Algorithm 2.2 Search for frequency of a pattern in PAT-tree

The advantages of PAT-trees are as follows: (1) They are easy to construct and maintain. (2) Any prefix sub-string and its frequency count can be found very quickly using a PAT-tree. (3) The space requirement for a PAT-tree is linear to the size of the input text.

3. Pat-tree with the deletion function

The block diagram of the PAT-tree with the deletion function is shown in figure 3.1.

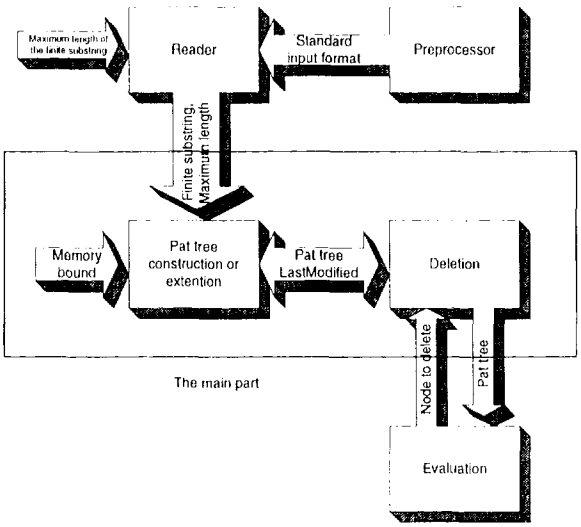

Fig. 3.1 The Block Diagram of PAT-tree Construction.

Implementing the deletion function requires two functions. One is the evaluation function that evaluates the data elements to find the least important element. The second function is release of the least important element from the PAT-tree and return of the freed node.

3.1 The Evaluation function

Due to the limited memory capacity of a PAT-tree, old and unimportant elements have to be identified and then deleted from the tree in order to accommodate new elements. Evaluation is based on the following two criteria: a) the oldness of the elements, and b) the importance of the elements. Evaluation of an element has to be balanced between these criteria. The oldness of an element is judged by how long the element has resided in the PAT-tree. It seems that a new field in each node of a PAT-tree is needed to store the time when the element was inserted. When the n-th element was inserted, the time was **n**. The resident element will become old when new elements are gradually inserted into the tree. However, old elements might become more and more important if they reoccur in the input text. The frequency count of an element is a simple criterion for measuring the importance of an

element. Of course, different importance measures can be employed, such as mutual information or conditional probability between a prefix and suffix. Nonetheless, the frequency count is a very simple and useful measurement.

To simplify the matter, a unified criterion is adopted. Under this criterion no additional storage is needed to register time. A time lapse will be delayed in order to revisit and evaluate a node, and hopefully, the frequency counts of important elements will be increased during the time lapse. It is implemented by way of a circular-like array of tree nodes. A PAT-tree will be constructed by inserting new elements. The insertion process takes a free node for each element from the array in the increasing order of their indexes until the array is exhausted. The deletion process will then be triggered. The evaluation process will scan the elements according to the array index sequence, which is different from the tree order, to find the least important element in the first k elements to delete. The freed node will be used to store the newly arriving element. The next position of the current deleted node will be the starting index of the next k nodes for evaluation. In this way, it is guaranteed that the minimal time lapse to visit the same node will be at least the size of the PAT-tree divided by k.

In section 4, we describe experiments carried out on the learning of high frequency word bi-grams. The above mentioned time lapse and the frequency measurement for importance were used as the evaluation criteria to determine the learning performance under different memory constraints.

3.2 The Deletion function

Deleting a node from a PAT-tree is a bit complicated since the proper structure of the PAT-tree has to be maintained after the deletion process. The pointers and the last decision node have to be modified. The deletion procedure is illustrated step by step by the example in Fig. 3.2. Suppose that the element in the node x has to be deleted, i.e. the node x has to be returned free. Hence, the last decision node y is no longer necessary since it is the last decision bit which makes the branch decision between DATA(x) and the strings in the left subtree of y. Therefore, DATA(x) and DECISION(y) can be removed, and the pointers have to be reset properly. In step 1, a) DATA(x) is replaced by DATA(y), b) the backward pointer in z pointing to y is replaced by x, and c) the pointer of the parent node of y which points to y is replaced by the left pointer of y. After step 1, the PAT-tree structure is properly reset. However the node y is deleted instead of x. This will not affect the searching of the PAT-tree, but it will damage the algorithm of the evaluation function to keep the time lapse properly. Therefore, the whole record of the data in x is copied to y, and is reset to the left pointer of the parent node of x be y in the step 2. Of course, it is not necessary to divide the deletion process into the above two steps. This is just for the sake of clear illustration. In the actual implementation, management of those pointers has to be handled carefully. Since there is no backward pointer which points to a parent decision node, the relevant nodes and their ancestor relations have to be accessed and retained after searching DATA(x) and DATA(y).

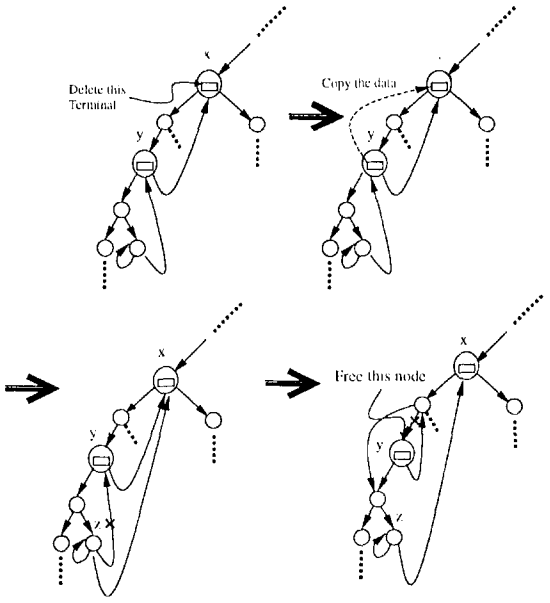

Fig. 3.2 The deletion process

4. Learning word collocations by Pat-trees

The following simple experiments were carried out in order to determine the learning performance of the PAT-tree under different memory constraints. We wanted to find out how the high frequency word bi-grams were retained when the total number of different word bi-grams much greater than the size of the PAT-tree.

4.1 The testing environment

We used the Sinica corpus as our testing data. The Sinica corpus is a 3,500,000-word Chinese corpus in which the words are delimited by blanks and tagged with their part-of-speeches[Chen 96]. To simplify the experimental process, the word length was limited to 4 characters. Those words that had more than four characters were truncated. A preprocessor, called reader, read the word bi-grams from the corpus sequentially and did the truncation. Then the reader fed the bi-grams to the construction process for the Pat-tree. There were 2172634 bigrams and 1180399 different bi-grams. Since the number of nodes in the PAT-trees was much less than the number of input bi-grams, the deletion process was carried out and some bi-grams was removed from the PAT-tree. The recall rates of each different frequency bi-grams under the different memory constraints were examined to determine how the PAT-tree performed with learning important information.

4.2 Experimental Results

Table 4.1 Finding the minimum of the next 200 nodes.

freq \ size	1/64	2/64	3/64	4/64	5/64	6/64	7/64	8/64
> 250	100	100	100	100	100	100	100	100
> 100	100	100	100	100	100	100	100	100
> 75	100	100	100	100	100	100	100	100
> 60	99.92	100	100	100	100	100	100	100
> 50	99.27	100	100	100	100	100	100	100
> 40	98.3	99.96	100	100	100	100	100	100
> 35	96.7	99.89	100	100	100	100	100	100
> 30	94.62	99.61	100	100	100	100	100	100
> 25	91.63	98.79	99.93	100	100	100	100	100
> 20	85.46	97.02	99.63	99.95	100	100	100	100
> 15	76.13	92.87	98.37	99.61	99.89	99.94	99.98	100
> 10	62.35	83.2	93.19	96.95	98.59	99.3	99.6	99.89
> 5	39.48	60.95	74.56	83.19	88.55	91.86	94.18	96.31
> 3	23.52	43.56	57.01	66.44	73.56	78.78	83.05	86.63
> 2	14.82	29.34	43.55	52.25	59.45	65.37	70.55	74.81
> 1	6.51	12.97	19.44	25.68	31.85	38.01	44.62	48.76

Different time lapses and PAT-tree sizes were tested to see how they performed by comparing the results with the ideal cases. The ideal cases were obtained using a procedure in which the input bi-grams were pre-sorted according to their frequency counts. The bi-grams were inserted in descending order of their frequencies. Each bi-gram was inserted n times, where n was its frequency. According to the deletion criterion, under such an ideal case, the PAT-tree will retain as many high frequency bi-grams as it can.

Table 4.2 Input bi-grams in descending order of their frequencies.

freq \ size	1/64	2/64	3/64	4/64	5/64	6/64	7/64	8/64
> 250	100	100	100	100	100	100	100	100
> 100	100	100	100	100	100	100	100	100
> 75	100	100	100	100	100	100	100	100
> 60	100	100	100	100	100	100	100	100
> 50	100	100	100	100	100	100	100	100
> 40	100	100	100	100	100	100	100	100
> 35	100	100	100	100	100	100	100	100
> 30	100	100	100	100	100	100	100	100
> 25	100	100	100	100	100	100	100	100
> 20	100	100	100	100	100	100	100	100
> 15	100	100	100	100	100	100	100	100
> 10	100	100	100	100	100	100	100	100
> 5	46.13	92.26	100	100	100	100	100	100
> 3	24	48	72	96	100	100	100	100
> 2	15	30	45	60	75	90	100	100
> 1	6.55	13.1	19.65	26.19	32.74	39.29	46.26	52.39

The deletion process worked as follows. A fixed number of nodes were checked starting from the last modified node, and the one with the minimal frequency was chosen for deletion. Since the pointer was moving forward along the index of the array, a time lapse was guaranteed to revisit a node. Hopefully the high frequency bi-grams would reoccur during the time lapse. Different forward steps, such as 100, 150, 200, 250, and 300, were tested, and the results show that deletion of the least important elements within 200 nodes led to the best result. However the performance results of different steps were not very different. Table 4.1 shows the testing results of step size 200 with different PAT-tree sizes. Table 4.2 shows the results under the ideal cases. Comparing the results between Table 4.1 and Table 4.2, it is seen that the recall rates of the important bi-grams under the normal learning process were satisfactory. Each row denotes the recall rates of a bi-gram greater than the frequency under different sizes of PAT-tree. For instance, the row 10 in Table 4.1 shows that the bi-grams which had the frequency greater than 20, were retained as follows: 85.46%, 97.02%, 99.63%, 99.95%, 100%, 100%, 100%, and 100%, when the size of the PAT-tree was 1/64, 2/64,..., 8/64 of the total number of the different bi-grams, respectively.

5. Conclusion

The most appealing features of the PAT-tree with

deletion are the efficient searching for patterns and its on-line learning property. It has the potential to be a good on-line training tool. Due to the fast growing WWW, the supply of electronic texts is almost unlimited and provides on-line training data for natural language processing. Following are a few possible applications of PAT-trees with deletion.

a) Learning of high frequency patterns by inputting unlimited amounts of patterns. The patterns might be character/word n-grams or collocations. Thus, new words can be extracted. The language model of variable length n-grams can be trained.
b) The most recently inserted patterns will be retained in the PAT-tree for a while as if it has a short term memory. Therefore, it can on-line adjust the language model to adapt to the current input text.
c) Multiple PAT-trees can be applied to learn the characteristic patterns of different domains or different style texts. They can be utilized as signatures for auto-classification of texts.

With the deletion mechanism, the memory limitation is reduced to some extent. The performance of the learning process also relies on the good evaluation criteria. Different applications require different evaluation criteria. Therefore, under the current PAT-tree system, the evaluation function is left open for user design.

Suffix search can be done through construction of a PAT-tree containing reverse text. Wildcard search can be done by traversing sub-trees. When a wildcard is encountered, an indefinite number of decision bits should be skipped.

To cope with the memory limitation on the core memory, secondary memory might be required. In order to speed up memory accessing, a PAT-tree can be split into a PAT-forest. Each time, only the top-level sub-tree and a demanded lower level PAT-tree will resided in the core memory. The lower level PAT-tree will be swapped according to demand.

References

de la Briandais, R. 1959. File searching using variable length keys. *AFIPS western JCC*, pp. 295-98, San Francisco. Calif.

Chen, Keh-Jiann, Chu-Ren Huang, Li-Ping Chang and Hui-Li Hsu. 1996. Sinica Corpus: Design Meghodology for Balanced Copra. *11th Pacific Asia Conference on Language, Information, and Computation (PACLIC II)*. pp. 167-176.

Flajolet, P. and R. Sedgewick. 1986. Digital search trees revisited. *SIAM J Computing*, 15;748-67.

Frakes, William B. and Ricardo Baeza-Yates. 1992. *Information Retrieval, Data Structures and Algorithms*. Prentice-Hall.

Fredkin, E. 1960. Trie memory. *CACM*. 3, 490-99.

Gonnet, G. 1987. PAT 3.1: An Efficient Text Searching System, User's Mannual. UW Centre for the New OED, University of Waterloo.

Hung, J. C. 1996. Dynamic Language Modeling for Mandarin Speech Retrieval for Home Page Information. Master thesis, National Taiwan University.

Morrison, D. 1968. PATRICIA-Practial Algorithm to Retrieve Information Coded in Alphanumeric. *JACM*, 15;514-34