Computers and Electrical Engineering xxx (2008) xxx–xxx

Contents lists available at ScienceDirect

Computers and Electrical Engineering

journal homepage: www.elsevier.com/locate/compeleceng

# DMP-tree: A dynamic M-way prefix tree data structure for strings matching

Nasser Yazdani*, Hossein Mohammadi

*Router Laboratory, ECE Department, University of Tehran, Tehran, Iran*

ARTICLE INFO

Available online xxxx

*Keywords:*
String matching
Prefix matching
Multi indexing
DMP-tree
IP Lookup

ABSTRACT

We propose DMP-tree, a dynamic M-way prefix tree, data structure for the string matching problem in general and prefix matching in particular. DMP-tree has been initially devised for fast and efficiently handling prefix matching which constitutes the building block of some applications in the computer realm and related area. It is assumed there are strings of an alphabet $\Sigma$ which are ordered. The data strings can have different lengths and some of them can be prefixes of others. Two well known applications of prefix matching are layers 3 and 4 switching in TCP/IP protocols. In layer 3 switching, routers forward an IP packet by checking its destination address and finding the longest matching prefix from a database. In layer 4 switching, the source and destination addresses are used to classify packets for differentiated service and Quality of Services (QoS). DMP-tree is a superset of B-tree. When none of the data strings are a prefix of each other, DMP-tree is the same as B-tree. In DMP-tree, no data string can be in a higher level than another data string which is its prefix. This requires a special procedure for node splitting. Indeed, node splitting differentiates DMP-tree from B-tree. The proposed data structure is simple, well defined, easy to implement in hardware or software and efficient in terms of memory usages and search time compared to other data structures proposed for prefix matching. We have implemented DMP-tree and the experimental results for simulated IP prefixes from the {0,1} character set show an average search time of $\mathrm{Log}_M^N$ for a large number of $N$, number of data elements, when the internal node branching factor $M$ is big enough ($\geqslant 5$).

© 2008 Elsevier Ltd. All rights reserved.

## 1. Introduction

Quickly and efficiently locating prefixes matching a query string is crucial in some applications. The prior assumption in these applications is that there are strings of an alphabet $\Sigma$ which are ordered. The strings can have different lengths and be prefixes of each other. The following lists a few real world applications as examples for which the efficient prefix matching is very critical:

1. Routers forward an IP packet by finding the next hop on the path towards the final destination. To perform this, a routing table database consisting of pairs of IP prefixes, or network addresses, and their corresponding next hop addresses is searched to find the longest IP prefix matching the packet destination address. Finding the next hop becomes harder as the increasing number of hosts on the Internet increases the size of routing tables. On the other hand, as communication line speeds increase, the time to process and forward a packet gets smaller. For instance, a router connected to

* Corresponding author.
*E-mail addresses:* yazdani@ut.ac.ir (N. Yazdani), hosm@ece.ut.ac.ir (H. Mohammadi).

0045-7906/$ - see front matter © 2008 Elsevier Ltd. All rights reserved.
doi:10.1016/j.compeleceng.2008.04.011

a 10 giga data line with a routing table of 164K IP prefixes has to find the next hop address in 50 ns. This problem is referred as layer 3 switching in the network community. The alphabet in this application is very limited, only $\{0,1\}$.

2. Internet Service Providers (ISPs) like to provide different services to different customers. Some organizations filter packets from the outside world by installing firewalls to deny access to unauthorized sources. Supporting these functionalities requires packet filtering or packet classification mechanisms in layer 4 of TCP/IP protocols. Forwarding engines must be able to identify the context of packets and classify them based on their source and destination addresses, protocols, etc., at wire speed. Routers handle this by keeping a set of rules which applies to ranges of network addresses in a database. Thus, we have to deal with the prefix matching problem in two dimensional space; i.e., for the source and destination addresses of a packet.
3. String processing has been an active research area from the dawn of computer science. Availability of extremely large amounts of textual data such as medical documents, libraries, archives, marketing information, etc., have motivated and influenced this study. The produced results and algorithms have found application in many diverse fields ranging from dynamic dictionary matching [1] and spell checking to molecular biology [13] and DNA matching. In all of these applications there are a set of patterns which can be dynamic. Besides insertion and deletion, a user is interested in finding all occurrences of a query string in the data set. The general substring search can be translated into the prefix matching problem as will be explained latter.
4. Other applications include directory lookups for telephones and social security numbers [24]. Ref. [2] relates the prefix matching problem to computer speech recognition by compactly encoding pronunciation prefix trees. A method to improve the parsing process of source codes which uses prefix matching has been introduced in [8]. The approach identifies the previously-parsed prefixes of a source, creates parsers in the parser states corresponding to the identified prefix and parses the remaining portion of the translation unit. Finally, we refer the reader to [12] which utilizes prefix matching in data compression which is crucial in database and data communication applications. The proposed approach there parses the input stream of data symbols into prefixes and data segments and uses the previously longest matching prefixes to compress the data.

The prefix matching search has been performed by the Trie structure [16]. A Trie is essentially an M-way tree where a branch in each node corresponds to a letter or character of alphabet $\Sigma$. A string is represented by a path from the root to a leaf node. The Trie structure has been modified and applied to most of the applications discussed above [24,21,22,19,10] The main problem with Trie is that it is inflexible; i.e. the number of branches corresponds to the number of characters, and keeps some extra nodes as place holders. Furthermore, in general, the search time is proportional to the length of the input strings. Others like Ferragina and Grossi [13] have tried to apply B-tree to the prefix matching problem. They have transferred the prefix matching query into the range query. It seems this approach does not work well in time critical applications such as IP lookup.

We propose DMP-tree as a new indexing and searching scheme for prefix matching. DMP-tree is a dynamic M-way tree which is built bottom up like B-tree. When data elements are disjoint, none of them is a prefix of other, the DMP-tree behaves exactly like B-tree. This is a nice property of our method. This is why we believe B-tree is a special case of DMP-tree. Our method can efficiently handle a wide range of prefix matching queries. We later explain how DMP-tree can be applied to the general pattern matching problem. However, in first step, besides the longest prefix match queries, we are also interested in exact match, smallest prefix match, all prefix match and listing all strings having prefix of a given pattern queries. Our solutions are efficient in memory usage and search time while scaling well to a larger data size or higher dimensions.

The rest of the paper is organized as follows. Section 2 explains the background of the work and addresses the basic issues. Section 3 discusses the DMP-tree structure. Application of DMP-tree to the general pattern matching query comes in Section 4. Section 5 discusses about general pattern matching. Section 6, contains our experimental results. We discuss related work in Section 6. Finally, Section 7 concludes the paper.

## 2. Background and basic issues

Tree structures keep the data elements sorted. Thus, in order to apply any tree structure to a data set, it is essential to have a mechanism to sort them. The traditional sort mechanism based on the value and lengths of strings [18] does not efficiently handle prefix matching queries and has to transfer them into range queries [13] of different lengths when the strings are prefixes of each others. The following definition gives a simple approach to compare and sort strings of different lengths. It is worth noting that we assume the characters in the alphabet are ordered.

**Definition 1.** Assume there are two strings $A = a_1a_2 \dots a_n$ and B = $b_1b_2 \dots b_m$, where $a_i$ and $b_j$ are characters of alphabet $\Sigma$ and there is a character $\perp$ which belongs to. $\Sigma$ Then

1. If $n = m$, the two strings have the same length and the values of $A$ and $B$ are compared to each other based on the order of characters in $\Sigma$.

2. If $n \neq m$ (assume $n < m$), the two substrings $a_1a_2 \ldots a_n$ and $b_1b_2 \ldots b_n$ are compared to each other. The substring with bigger (smaller) value is considered bigger (smaller) if the two substrings are not equal. If $a_1a_2 \ldots a_n$ and $b_1b_2 \ldots b_n$ are equal, then, the $(n+1)$th character of string $B$ is checked. $B \leqslant A$ if $b_{n+1}$ is equal or before $\perp$ in the ordering of characters in $\Sigma$, and $B > A$ otherwise.

The $\perp$ character should be chosen in such a way that the probability of any character(s) in the lower order or upper order $\perp$ of is roughly equal. For instance, in the English alphabet, assuming the probability of a character to be in the range $A$ –$M$ or $N$–$Z$ in a text to be roughly 50%, $M$ can be considered as $\perp$. Then BOAT is smaller than GOAT and SAD is bigger BALLOON. CAT is considered bigger than CATEGORY since the fourth character in CATEGORY, E, is smaller than M. In the binary alphabet, {0,1}, assuming $\perp$ is 0, clearly, 1101 is greater than 1011 and smaller than 11101, and 1011 is greater than 101101. Therefore, Definition 1 gives us the necessary tool to compare and determine the relative position of all strings with respect to each other.

**Definition 2.** Assume there are two strings $A = a_1a_2 \ldots a_n$ and $B = b_1b_2 \ldots b_m$, where $a_i$ and $b_j$ are characters of alphabet $\Sigma$. Then $A$ and $B$ are matching if $n = m$ and two strings are identical, or (assuming $m > n$, two $a_1a_2 \ldots a_n$ and $b_1b_2 \ldots b_n$ substrings are the same. Otherwise, $A$ and $B$ do not match.

**Definition 3.** Two strings $A$ and $B$ are disjoint if each is not a prefix (or substring) of the other.

**Definition 4.** A string $S$ is called an *enclosure* if there exists at least one data string $A$ such that $S$ is a prefix of $A$. $A$ is called an *enclosed* data element.

For example, *BAT* and *PHONE* are disjoint, but *DATE* is an enclosure of *DATED* and *DATELINE*. As another example, 1011 is an enclosure of 1011010. We call these elements enclosures since they include other data strings in their spaces. An enclosure represents a data range as a point in the data space. For instance, all data strings which are included in the range of 1011, such as 1011001, 1011010, and 1011 itself are considered as a point represented by 1011. A data element may be included in an enclosure or be disjoint with all other elements.

We have proposed a binary prefix tree for prefix matching in [25,31] which utilizes Definition 1 and the idea of enclosure. It takes a prefix and all data elements enclosed in its space as a data point represented by the prefix. This process is called *enclosurizing*. After the enclosurizing process, virtually all enclosed data elements are removed from the list. Thus, the remaining data elements are disjoint. They can be sorted and a binary search tree can be built. If the split point was an enclosure, the enclosure prefix is put in the root and the enclosed elements in its space are distributed in the subtree. To build the binary prefix tree, the data elements are sorted and enclosurized first. Refs. [25,31] proposed two approaches to sort and enclosurize data elements based on bubble and quick sort.

## 3. DMP-tree

Clearly the binary tree structure is not efficient in search. The worst case search time for the binary prefix tree proposed in [25,31] is *O(W)* where *W* is the maximum length of the data elements. Furthermore, new insertion and deletion may deteriorate the index structure and the search process. DMP-tree (Dynamic M-way Prefix Tree) is proposed to solve both of these shortcomings. DMP-tree is similar to B-tree in general, but with the following distinctions.

1. No data element can be in a higher level than its enclosure in the index tree structure.
2. DMP-tree does not guarantee minimum node utilization.
3. It is not possible to guarantee that the final tree is balanced.

The first property differentiates DMP-tree from the binary prefix tree in [25,31]. Indeed, in the binary prefix tree, enclosures are always in the higher levels than their corresponding enclosed elements. In DMP-tree an enclosure can be in the same level with its enclosed data. Nevertheless, no enclosed element can be in a higher level than its enclosure. The second and third properties come from the node splitting policy discussed in the next subsection. Our experimental results show these properties fade when the branching factor in the internal nodes is large and the data set is big. Indeed, in this case DMP-tree approaches B-tree in terms of search and memory utilization. In the following, we formally define node splitting, insertion, space division, merging and search mechanisms. Other procedures such as building the tree or deletion are the same as B-tree [4].

### 3.1. Node splitting

The main problem in node splitting is determining the spilt point. In *B*, the median element is chosen to split the space. Identifying the median is easy. However, in the dynamic M-way prefix tree, finding the split point is complicated since the tree must satisfy the condition that no data element can stay in a higher level than its enclosure. The split point can be selected in the following order for an overflowed node:

1. If all strings in the node are disjoint, the *median* is selected as the spilt point.
2. If there is an *enclosure* which encloses all other data strings in the node, it is selected for splitting the tree node.
3. In the cases not mentioned above, the data elements can be a combination of disjoint spaces and strings. In this case, an enclosure or a disjoint element which gives the best splitting result is chosen as the split point. We must avoid taking elements in the left most or right most as the split point as much as possible since they create the worst splitting scheme, i.e., one node full and another empty.

It is always possible to split a node. The worst case is when we have to choose the element in the beginning or at the end as a split candidate. This may result to a larger tree height and poor search performance.

Node splitting here is the same as splitting data elements in two parts to build the binary prefix tree in [25,31]. The data element in the node must be enclosurized and sorted. Then the median can be chosen as the split point. If the data elements are disjoint, we will use the first policy. If one prefix encloses all others, we will end up with the second policy. Otherwise, the node is split based on the third policy. Refs. [25,31] introduces two approaches for enclosurizing and sorting. A problem with both proposed methods is that they do not guarantee the best splitting result. The best splitting result here means to evenly distribute the data elements in two spaces as long as the DMP-tree properties are not violated. Here, we give an algorithm which guarantees to find the best possible splitting in a node. The algorithm assumes the previous data elements in the node are sorted and the place of the new inserted data string has been already found. It keeps a bit for each element which shows whether it is enclosed in a data space or not. Clearly, a data element enclosed in a data space cannot be a split candidate.

```
/* node is a pointer to the full bucket. SplitPoint is the element should go up*/
Split (bucket *node,element *new_element, element *SplitPoint)
{
    Insert new_element into node;
    /* now node has one extra element and we should select the split point*/
    /* the first step is enclosurizing*/
    For each P in node do
      if P has any enclosure in node then
        Mark P as enclosed;
    SplitPoint = Middle of enclosures in node;
}
```

It is easy to prove the algorithm correctly enclosurizes the data elements. If an element is already identified as enclosed we do not need to check it again. But what if there was an element in the list which is in the space of the enclosed element? Fortunately, this will not create any problems since that element will also be in the data space of the prefix which is enclosure of the element under check. That is why we can terminate testing for any element which is enclosed in another element data space. Finally, the prefix which encloses all data elements will be compared with all of them.

The best case running time of the split algorithm is $O(N)$ when the first prefix encloses all others. The worst case is $O(N^2)$ when data elements are disjoint.

## *3.2. Space division*

Space division is encountered when a data string is replaced by its enclosure in an internal node. The data strings in a subtree are sorted and the data space is divided by the data elements in the root node of the subtree. According to the tree definition, all strings in the left subtree are smaller than the root and the elements in the right subtree are bigger. When the root is replaced by another data element, it is impossible to guarantee the resulted tree satisfies this condition. Therefore, we need to check all elements in the subtree and resort elements which violate this condition. In other words, we need to divide the data space with respect to the new string in the root. However, we need to divide the subspace either in the left or right. This is due to the fact that if the new string is bigger than the replaced string in the root, all elements in the left subtree will be smaller than the new string and will remain in their original place. However, some data elements in the right subtree may be smaller and must be moved to the left. This must be done recursively from the top to the leaves. Since the elements are sorted this can be done faster. In the following, we give a formal procedure for dividing a (sub)space with respect to a new string. We have assumed the enclosure or the new string is bigger than the original one. The elements in the right subtree are checked to see if they are smaller.

Checking the left subtree is the same except that the less than sign (<) in the while condition must be replaced with the greater than sign (>).

```
/*node is a pointer to a bucket and str is the new split element. left and right are pointers of str */
SpaceDiv(bucket *node,element *str,*left,*right)
{
    if node is null then return;
    for(i = first element of node; i < str && i ≠ nil; i = next element)
    {
      delete i from right;
      add i to left;
    }
    SpaceDiv(i.left,str,newleft,newright);
    add newleft to end of left;
    add newright to beginning of right;
}
```

It can be shown that this procedure can be avoided in the building process if strings with shorter lengths are inserted first. The reason is that enclosures always have shorter lengths and if they are added first, we will never face a situation such that the new inserted element is an enclosure of others forcing us to divide a space in the index tree.

## 3.3. Merge

After space division, the new elements which violate sort condition in the tree must be moved to the other side of the element in the root of the subtree. This is how the merge operation is encountered. Generally speaking, the merge operation takes two sub-trees and merges them into one subtree. The operation is recursive starting from the root of the subtrees. The data elements in the root of the subtrees are put in one node. If the node has been overflowed, it is split and the split point is inserted in the upper node as discussed in the next subsection.

```
/* leftTree and rightTree are pointers to subtrees to be merged. left and right are pointers to the
left and right node and splitStr is the split point. */
Merge(leftTree; rightT ree; splitStr; left; right)
    if leftTree is NULL, then, return rightTree
    if rightTree is NULL, then, return leftTree
    /* store the last pointer in leftTree node and first in the rightTree */
    leftTmp = the last pointer in leftTree;
    rightTmp = the first pointer in rightTree;
    if data elements in leftTree and rightTree can fit in a node,then,
        Move data elements in rightTree into leftTree.
        return leftTree as the result.
    else
        leftTree = leftTree & rightTree /* put all in one place */
        Split (leftTree; splitStr; left; right)
        Insert splitStr with left and right pointer in the upper node.
        Merge(leftTmp;rightTmp;splitStr;left; right);
end Merge;
```

## 3.4. Insertion

As B-tree, Insertion is the most important part of the building process of DMP-tree. It uses Split, SpaceDiv, Merge and NewNode functions to split a node if it is full, divides a space if an element is replaced by its enclosure, and merges two subtrees if necessary. Finally, it allocates a new node if a node is full and is going to be split. In formally defining the insertion procedure, we ignore some details which are important in the actual implementation. Right-Child and left-Child are simple macros which return the right and left pointers of a data element in an internal node.

```
/*tree is a pointer to the root of the index tree.*/
Insertion(tree; str)
    if tree is leaf, then,
        if tree is full, then,
            node = NewNode();
            Split(tree; node; str; s);
            insert s in the parent of tree.
    else
        insert str in tree.
        return;
    if str is enclosure of any element in the node pointed by tree, then,
        replace the closet contained element i in str with str;
        if (str > i)then,
                SpaceDiv(rightChild(str); str; left; right);
                Merge(leftChild(str); left);
        else
                SpaceDiv(leftChild(str); str; left; right);
                Merge(right, rightChild(str);
        insert i in tree
        return;
        i = first string in tree(node);
        while(i < str or no next element in tree(node))
                i = next string in tree;
        if str is greater than all elments (i is the last one), then;
                Insertion(rightChild(i); str)
        else
                Insertion(leftChild(i); str)
end Insertion;
```

The running time of the insertion procedure is *O(h)* where *h* is the height of the tree. However, inserting an enclosure when it causes space division takes more time since the SpaceDiv procedure has to be called to divide the space according to the new root element. The running time of this process is also proportional to the height of the tree.

**Example.** We illustrate the process of building DMP-tree for the data elements of Table 1. The data set are from the binary alphabet {0,1} which is the most important application of DMP-tree. The table contains 34 strings and data elements longer than 5 bits have an abbreviation to make the final representation of the tree easier. It is assumed that each node, internal or leaf has at most four data elements, implying each space is divided by five. The data elements are inserted randomly. Initially, the tree is empty. Then 01011, 1011010, 10110001 and 0100110 are added to the root node. Adding 110 causes overflow and the node must be split. Since all data elements are disjoint, the median, 10110001, is chosen as split point. The tree after splitting is illustrated below.

$$\begin{array}{c} (\swarrow 10110001 \searrow) \\ (0100110, 01011) \quad (1011010, 110) \end{array}$$

Later, 10110011, 1101110010 and 00010 are inserted. The first two are added to the right leaf node and the last one to the left. Inserting 1011001 causes overflow on the right leaf and it must be split. Since 1011001 is an enclosure of 10110011, it can be a candidate as split point. Unfortunately, this element gives the worst splitting result since one space is empty and another full. However, since the other elements are disjoint, choosing the median, 1011010, gives a better result. Below is the tree obtained from splitting.

$$\begin{array}{c} (\swarrow 10110001 \quad \downarrow \quad 1011010 \searrow) \\ (00010, 0100110, 01011)(1011001, 10110011)(110, 1101110010) \end{array}$$

In the next step, inserting 01 and 10001101 causes overflow in the left leaf. 01 is enclosure for 0100110 and 01011, and can be a split point. Even though this does not give a good splitting result, it is the only viable candidate since 0100110 and

**Table 1**
A Sample data set of strings in{0,1} Alphabet

| String | Abbreviation | String | Abbreviation |
|---|---|---|---|
| 10 | - | 1101110010 | K |
| 1 | - | 10001101 | L |
| 110 | - | 11101101 | M |
| 1011 | - | 1010110 | N |
| 0001 | - | 100101 | O |
| 1011 | - | 100110100 | *P* |
| 10 | - | 101011011 | Q |
| 001100 | A | 11101110 | R |
| 1011001 | B | 10110111 | S |
| 1011010 | C | 011010 | T |
| 100110 | D | 011011 | U |
| 1001100 | E | 011101 | V |
| 10110011 | F | 0110010 | W |
| 10110001 | G | 101101000 | X |
| 1011001 | H | 101101110 | Y |
| 1011 | I | 00011101 | Z |
| 111010 | J | 011110110 | II |

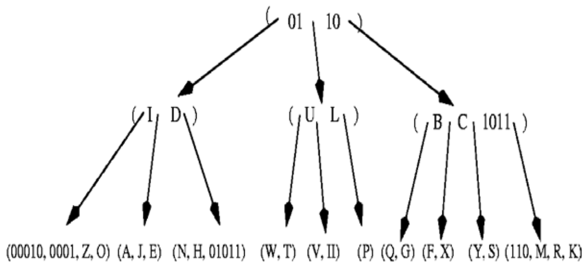

**Fig. 1.** An M-way tree for the prefixes in Table 1.

01011 cannot go to a higher level than 01 according to the prefix tree specification. Choosing one of the two other elements for splitting would make the situation worse.

$$\begin{array}{c}(\swarrow 01 \quad \downarrow \quad 10110001 \downarrow \quad 1011010 \searrow)\\ (00010, 0100110, 01011)(10001101)(1011001, 10110011)(110, 1101110010)\end{array}$$

Later adding 0001, 10110111, 11101101, 100110100,101011011,101101110 and 101101000 causes no specific issue. Next 1011 is inserted. 1011 is an enclosure of 1011010 and 10110001 at the first level. Clearly, 1011 cannot be in a lower level than its enclosed elements. Therefore, it must be added to a higher level or at least to the same level of its enclosed elements, i.e., level one. The elements in the lower level have been represented by their abbreviation from Table 1.

$$\begin{array}{c}\swarrow 01 \quad \downarrow \quad 10110001 \downarrow \quad 1011010 \downarrow \quad 1011 \searrow\\ (00010, 000110, D, 01011)(L, P, Q)(B, F, X)(S)(110, M, K)\end{array}$$

Fig. 1 shows the final tree structure.

## 4. Query processing

For the Longest Prefix Matching (LPM) of a query string, [25,31] proposes a method which exhaustively compares all elements in the nodes of the search path. This is due to the fact that an enclosure and its contained elements may be in the same level. DMP-tree allows this in order to reduce the height of the tree and consequently, the search time. Nevertheless, the following lemma states that this is not necessary.

**Lemma 1.** *Assume P is a prefix of query string Q in a node of DMP-tree. Then it is necessary and sufficient to compare Q up to the first data element in the node which is disjoint with P in order to find the Longest Prefix Matching (LPM) of Q.*

**Proof.** The proof comes directly from the sort property n the tree structure. All data elements in any node of DMP-tree are sorted based on definition 1. The search process tries to find the position of Q first. Clearly, the position of Q will be close to LPM. There are two cases. Either Q is bigger than LPM or smaller. In the first case we have to check the next element, S, in the node. If S is bigger than LPM then, it must be bigger than Q. The reason is simple. If S is disjoint with LPM, it must be bigger than Q. If S matches LPM, there two cases. First, its length is smaller and it is a prefix of LPM and consequently a prefix of Q. Since LPM is the longest prefix, then, LPM and Q have to have the same order, smaller or bigger and here bigger, compared to S. This implies $S > Q$. In the second case, S is longer than LPM. Definitely, $S$ cannot match Q. Otherwise; it will be the longest prefix matching Q rather than the previous element. □

Using this lemma may not help much when the number of data elements is small. However, when the number of elements to compare is large, the reduction in matching time may be considerable. Before giving a formal description of the search procedures, we introduce a property of DMP-tree which simplifies and expedites the search process.

**Lemma 2.** *If there are two matching prefixes of a query string which are at two different levels, the one in the lower level has longer length.*

**Proof.** Assume $A$ and $B$ are two prefixes of query string S in a DMP-tree index structure and $A$ is shorter and in a lower level than $B$. We will show that this is impossible. Since $A$ is shorter, it is an enclosure of $B$. If the index tree is built statically, $B$ will be put in the bag of $A$ and according to the Build-Tree procedure introduced in the previous section, $B$ will be pushed into a lower level node or at the same level as $A$.

This contradicts our prior assumption. If the tree is built dynamically, the data elements are inserted one by one into the index tree, and if $A$ is in the lower level this leads to a contradiction. Assume $A$ has been inserted first and $B$ has been added later. Since the data elements are sorted, $B$ will be close to $A$ and will be mapped to its space. If $A$ is in an internal node, there are different cases to consider. (i ) If $A$ is the only prefix of S in the node, $B$ will be added to the subtree rooted on $A$ and will be in a lower level or at most at the same level with $A$ if $B$ pops up from node splitting in the lower levels. (ii) There are different prefixes of S in the node but all of them are shorter than B. Then B will be added to the subtree rooted on the longest prefix of S in that node and consequently, it will be in a lower level or at most at the same level with A. (iii) there are different prefixes of S in the node but not all of them are shorter than B. In this case, $B$ will replace one of them and will be in the same level with A. if there was any split in the node, $B$ can not go higher than $A$ since it is shorter. If $A$ is in a leaf node, $B$ will also be added beside $A$ since it has to be in A's space. In the node splitting $A$ must come to the upper level before $B$ according to the splitting strategy. This also leads to a contradiction.

Assuming that $B$ is inserted first also results in a contradiction. According to the sort property of the tree, $A$ will map to a location and will represent a space to which $B$ belongs. If $B$ is in a leaf node, there are two cases: First, the number of elements in the space is small and all strings reside in a leaf node. In this case, $A$ will be added to this node and $B$ can not go to a level higher than A. In the second case, the number is big and the space has already been split. The split point belongs to the space and is in the upper level. Then in the insertion process of A, the search process will find the split point string and it will be replaced by $A$ according to the insertion procedure. This implies that $A$ cannot be in a lower level than B. □

With this lemma in mind, we give a formal procedure for searching the longest prefix of a given query string. It is worth noting that we do include Lemma 1 in this procedure since it is assumed the number of elements in each node of DMP-tree is limited.

```
{/* tree is a pointer to the root of the index tree and str is the query string.*/
LPMSearch (tree; str)
      if tree = NIL,return NULL;
      i = first element in tree(node).
      while(str<= i & i is not nil) do
        i = next element in tree(node).
        j = i;
      if i is not nil, then
        prefix = LPMSearch(leftChild(i); str).
      else
        prefix = LPMSearch(rightChild(j); str).
      if prefix is NULL, then,
        prefix = the longest prefix in tree(Node) matching str.
      return prefix;
end Search;
```

**Theorem 1.** *The search procedure given above correctly finds the longest matching prefix in DMP-tree tree.*

**Proof.** The correctness of the last part of the procedure comes from Lemma 2 which states the prefixes in the lower level are longer. However, it has to be shown that by following one branch, we do not miss anything. Assume $A$ is a prefix of query string S which is not in the path of nodes followed by the search process. Since $A$ is an enclosure of S, the search process must map S to the space of A. This is due to the fact that search and sort both try to map or locate a data element position with respect to others. There are two possibilities. First, the number of data elements in the space of *A* is few and all of them reside in a leaf node. In this case, S will hit this leaf node and will find A. Another possibility is that the data elements are many and the space has been split. Since *A* is one of the split points, it is an enclosure, and the search process will hit A. Either *A* is in a higher level than the enclosed strings in its space or it is in the same level with them. The first case implies the enclosed strings are in the subtree rooted at A. Thus, the search process definitely will visit A. In the second case, A and its enclosed elements must be in a node and close to each other according to the sort property. Otherwise, the node will be split and A will go to the upper level, which is the same as the first case. Since the search process exhaustively checks all elements in the node, it will definitely meet A. This contradicts the prior assumption. □

The LPM-Search procedure can be easily modified to find the shortest prefix matching a query string. The main difference between two queries processes is that we do not need to process the tree structure from the root to the leaves in this case. The search process can stop in any level in which at least one matching prefix has been found. Correctness of the procedure comes from Lemma 2.

```
/* tree is a pointer to the root of the index tree and str is the query string.*/
SPMSearch(tree; str)
    if tree = NIL, then return NULL;
    prefix = the shortest prefix in tree(Node) matching str.
    if prefix is not NULL, then, return prefix;
    i = first element in tree(node).
    while (str< = i && i is not nil) do
      i = next element in tree(node).
      j = i;
      if i is not nil, then
            prefix = SPMSearch(leftChild(i); str).
      else
            prefix = SPMSearch(rightChild(j); str).
            return prefix;
end Search;
```

Finding all prefixes of a given query string is almost the same as finding the longest and shortest prefixes. The only difference is to report any matching prefixes during the search process. We examine the procedure below and prove the algorithm does not miss any matching prefix.

```
/* tree is a pointer to the root of the index tree and str is the query string.*/
PrefixSearch(tree; str)
      if tree = NIL, then return NULL;
      print all prefixes in tree(Node) matching str.
      k = first element in tree(node).
      while(str< = k && k is not nil) do
            k = next element in tree(node).
      if k is not nil, then
            PrefixSearch(leftChild(k); str).
      else
            PrefixSearch(rightChild(j); str).
end Search;
```

It is worth noting that the search only follows one path even though there might be more than one prefixes matching the query string. This is a nice property of the algorithm. It shortens the search time.

**Theorem 2.** *The Prefix-Search procedure correctly finds all prefixes of a query string in a DMP-tree index structure.*

**Proof.** It is sufficient to show the algorithm does not miss any matching prefix. Starting from the root, the query string, say Q, will follow the same search path with its prefixes. In the search path, if we do not meet any matching prefixes, all the split points are disjoint with Q and according to the node splitting process they can not contain any prefix matching Q. Based on our definition of DMP-tree, prefixes of a string create a hierarchical space, starting with the shortest in the root, like shells of an onion. The query string has to fall inside this space in order to match any of them. Theorem 1 guarantees the process to find the longest prefix does not miss any matching prefix. Since the process to find the longest matching prefix and finding all prefixes matching $Q$ are identical when we have not encountered any matching prefix, then, we have not missed any of them. When the procedure ends up with some matching prefixes in the search path, it will print all of them. Matching prefixes create data spaces which potentially can contain matching prefixes of $Q$. However, it is sufficient to only follow the longest prefix subtree. The reason is simple. Let us assume one of the other prefixes has a prefix, say A, of $Q$ in its subtree. $A$ can be shorter than the Longest Prefix Matching (LPM), or longer. Assuming $A$ is shorter than LPM contradict Lemma 2 which states that the prefixes in the lower level are longer. If $A$ is longer than the LPM, then it has to be in the LPM's space and the search process will find it. All prefixes longer than LPM will be in the same order with $Q$ respect to the LPM, so tracing only one branch is sufficient. □

Finding all strings having prefix $Q$ is the last query addressed in this subsection. This query is the base of the general pattern matching algorithm discussed later in this paper. We define *match(P,Q)* function as the following:

$$match(P,Q) = \begin{cases} -1 & \text{if} \quad P \text{ and } Q \text{ are disjoint and } P > Q \\ 0 & \text{if} \quad P \text{ is a prefix } \ Q \\ 1 & \text{if} \quad P \text{ and } Q \text{ are disjoint and } P < Q \end{cases}$$

We also define a bit map, Searched. Each bit of Searched corresponds to a subtree pointed to by a pointer in an internal node and shows if the subtree has been searched. This bitmap is set to zero initially.

```
/* tree is a pointer to the root of the index tree and Q is the query string.*/
AllStringSearch (tree;Q)
    if tree = NIL, then return NULL;
    Set all bits of Searched to zero.
    K = first element in tree(node).
    switch = match(Q;K).
    while(switch<>1k is not nil) do
        J = K;
        if switch = 0/* they match. */print K.
            print K
        if corresponding bit in Searched to leftChild(K) is zero,
            AllStringSearch(leftChild(K); Q).
            set the corresponding bit in Searched to (leftChild(K) to 1.
    if corresponding bit in Searched to rightChild(K) is zero,
        AllStringSearch(rightChild(K); Q).
        set the corresponding bit in Searched to (rightChild(K) to 1.
    K = next element in tree(node).
    switch = match(Q;K).
end while;
if K is not nil and the corresponding bit in Searched to (leftChild(K) is zero,
        AllStringSearch (leftChild(k);Q).
if K is nil and the last bit in Searched is 0
AllStringSearch(rightChild(j);Q).
end Search;
```

Instead of linear search in each node, we can do binary search starting from the middle element. If the middle element matched, `match (Q,K)` returned zero, and the search has to check the left and right elements also. Otherwise, it just follow regular binary search procedure.

**Theorem 3.** *The All-String-Search procedure correctly finds all strings having a query string as their prefix.*

## 5. General pattern matching

Ref. [13] tries to solve the problem of finding all strings having prefix $P$. It uses a modified version of B-tree with the traditional sort mechanism for strings which is based on the string lengths. For instance, DATE is smaller than DATED since the first section of both string, DATE, is the same and the length of DATED is bigger. Based on this sort function, the strings having prefix P occupy a contiguous part in the tree structure.

Therefore, we only need to retrieve the leftmost and rightmost strings whose prefix is *P*. However, based on Manber and Myers [18] the leftmost string is adjacent to *P*'s position in the tree. The B-tree like index structure keeps pointers to the location of the data strings. Finding the location of P is costly in term of disk access. The method keeps a Patricia Trie [14] to reduce the number of disk access. Fig. 2 shows an example for {*ace, aid, atlas, atom, attenuate, by, bye, car, cod, dog, fit, lid, patent, sun, zoo*} data set. The strings are stored in the leaves by pointers to their locations in the secondary memory. Internal nodes keep the leftmost and rightmost pointers to subtrees in the lower levels.

DMP-tree can answer this query in a more efficient way. Fig. 3 illustrates the DMP-tree index structure for the same data set. As expected, the final tree is much more compact. There is no repetition of pointers. Indeed, DMP-tree gives much better memory utilization compared to the B-tree like structure proposed in [13]. To search all strings having 'at' prefix in the DMP-tree of Fig. 3, we apply the All-String-Search procedure discussed in the previous section. In the root, the data element 'atom', pointed by 5, is matching. We should follow the left and right pointer of 'atom' to find other matching elements. Assuming the data elements are random and there is no much correlation between them, we can safely assume the final tree is resemble to B-tree with the average height $\mathrm{Log}_N^M$, where *N* is the number of data and *M* is the branching factor in the internal node. The search should take $\theta\mathrm{Log}_N^M$, where *K* is a constant value which depends on the distribution of data. We expect $1 < K < 2$ for real pattern matching applications.

Unfortunately, for each node, we have to have $M - 1$ disk accesses to get the actual value of strings to compare. This makes the search process very costly. Ref. [13] proposes binary search in the nodes first and finally plugging a *Patricia Trie* for each node in order to reduce the disk access for each node to 1. We may utilize all of those techniques here. Furthermore, applying the following techniques reduces disk access and makes the search process faster.

- Beside pointers, to keep the first few characters of data elements in the tree nodes like [5]. This takes more space, but, It does not need other disk accesses for matching most of the time.
- To do binary search in the first nodes such as root. If there is a match, we can start from the last element to check for the subtree in the left and from the first element for the subtree in the right. The matching can stop when any element does not match.

The general substring search, finding all occurrences of a pattern *P* in a text $\theta$, can be solved by extending the prefix matching algorithm. To do this, the same technique in [13,1] is used. The suffix set of $\theta$ is defined as $\{\theta[i,|\delta|]\}$ where $1 \leqslant i \leqslant |\delta|$.

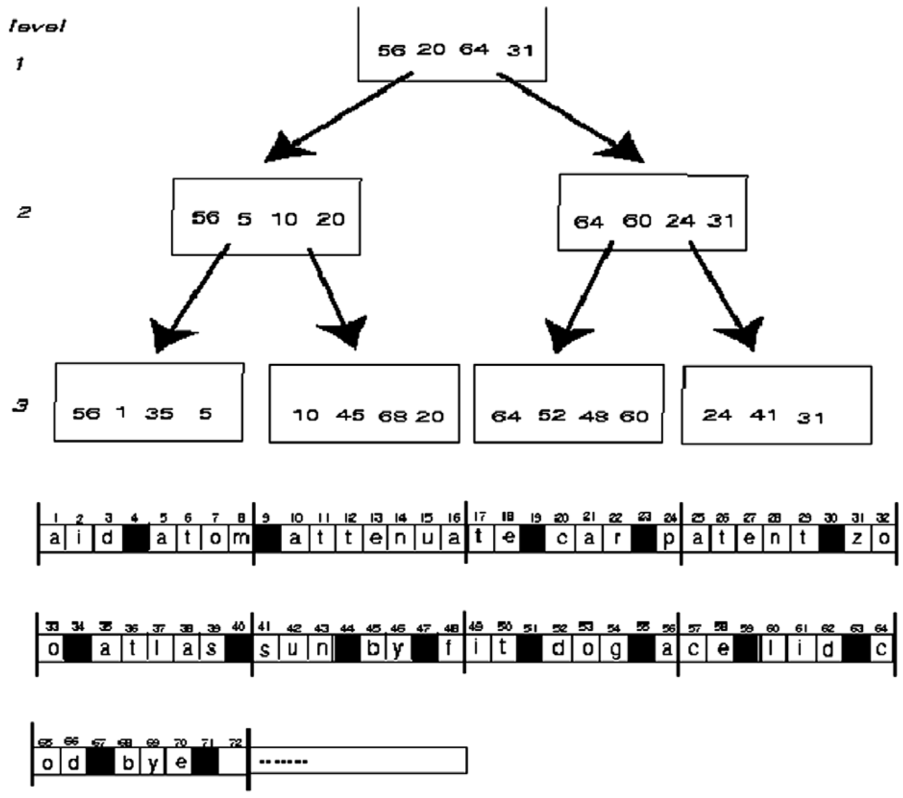
level
1
2
3
56 20 64 31
56 5 10 20
64 60 24 31
56 1 35 5
10 45 68 20
64 52 48 60
24 41 31
a i d a t o m a t t e n u a t e c a r p a t e n t z o
o a t l a s s u n b y f i t d o g a c e l i d c
o d b y e
-------

**Fig. 2.** An example of B-tree like index structure for prefix matching proposed in [13] for a sample data set.

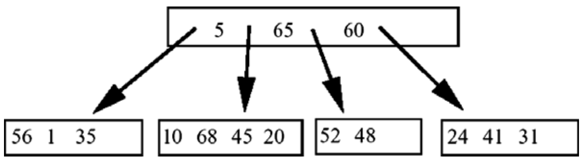

**Fig. 3.** A DMP-tree for the same data set in Fig. 2.

Then the problem is to find all strings having prefix P from the suffix set. For example, assuming *'attenuate'* as a text, its suffix set will be {*'e', 'te', 'ate', 'uate', 'nuate', 'enuate', 'tenuate', 'ttenuate', 'attenuate'*}. Assuming *P* is 'at', there are two occurrences in the text which can be found in {*'attenuate'*} and {*'ate'*} suffixes.

**Lemma 3.** *Assume a text $\theta$ with length N, a DMP-tree built for the suffix set of $\theta$ has the maximum height $\lceil \frac{N}{M-1} \rfloor$ where M is the maximum branching factor in the internal nodes.*

**Proof.** According to the property of DMP-tree, the worst case is when the elements of the suffix set are prefixes of each other. This is the case if $\theta$ consists of only one character. Since there are N elements in the suffix set, and each node, including leaves, can contain $M-1$ data element, the maximum height of tree will be $\lceil \frac{N}{M-1} \rfloor$. □

## 6. Experimental results

### 6.1. Binary alphabet results

DMP-tree is a superset of B-tree and when all data elements are disjoint, it acts exactly the same as B-tree. Therefore, we expect the average search and update time to be proportional to $\text{Log}_N^M$, where N is the number of data elements and M is the branching factor in the internal node. The memory utilization and the average search time of B-tree are well known [4,9]. All those results directly apply to DMP-tree. Even though this is the best case, our experimental results show this is the same as the average case for some applications such as IP lookup.

We do not have analytical results for the average and worst cases of DMP-tree. However, in some applications such as dictionary checking and IP lookup, when the data set is large and data elements are expected to be random and branching factor in internal nodes is large enough, the DMP-tree memory utilization and search performance approach B-tree. For instance, considering {0, 1} alphabet and the address lookup table application, the worst case happen when all $1^*$, $11^*$,... combination of IP prefixes are in the table and inserted first to the index tree. This creates a tree of one branching and adding any new element probably adds to the height of the tree. Let us assume the longest string has length *L* and *M* is the branching factor. According to Lemma 3 the initial height of the tree before inserting other prefixes will be $\lceil \frac{L}{M-1} \rfloor$ assuming the number of data elements in leaves and internal node are the same.

Let us see how important this is for the IP lookup application. IP prefixes used for routing have 8–32 bit lengths. Assuming the data set has all the worst case data and they are inserted first, the initial DMP-tree will have the height $\lceil \frac{32-8}{M-1} \rfloor$. Therefore, the branching factor *M* and the number of data $\lceil \frac{24}{M-1} \rfloor$ elements *N* determine the final height of the tree. For large value of *M* ($\geqslant$6) and *N* (N $\geqslant$ 60K), the initial heights for the worst case will be small (<5) compare to the final height. Since we have already included the worst case data, the remaining data elements will be disjoint with a high probability. This implies the final tree approach B-tree. To give an example, assume *M* = 9 and *N* = 64K. We expect the initial worst case height to be 3 and the final height of the tree to be $\text{Log}_9^{64K} = 5.1$. This exactly what our experiments shows.

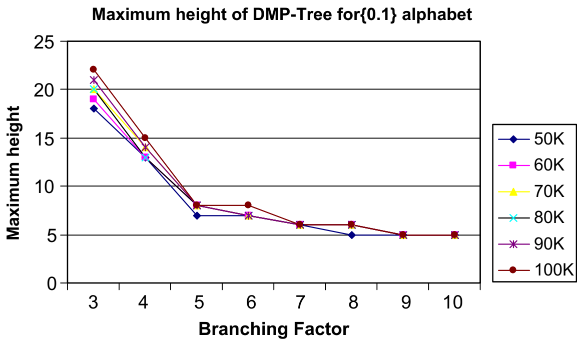

**Fig. 4.** The maximum heights of DMP-tree for the data sample of IP address prefixes.

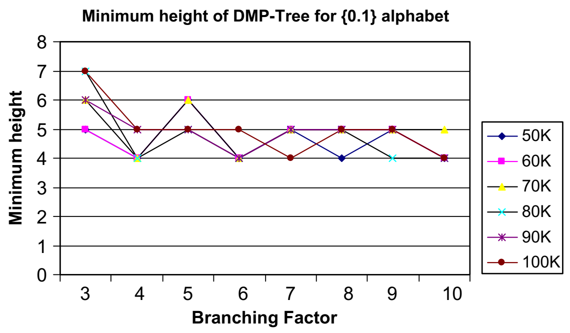

**Fig. 5.** The minimum heights of DMP-tree for the data sample of IP address prefixes.

We have implemented DMP-tree in a SUN SPARC 10 platform. To test the search performance and memory utilization with the IP lookup data, which is the hottest application of our method, some simulated data was produced and a few small modules for bit handling were included in the code. Six independent data sets 50K, ~60K,. . .,100K were produced. were produced. There were no duplications in the data sets. We ran each data set with different branching factors ranging from 3 to 10 and recorded the minimum, average and maximum heights. Figs. 4 and 5 show the maximum and minimum tree heights for each data set, respectively. As these figures indicate, the difference between maximum and minimum values for the same data decrease with increasing *M*. For small values of *M*, the difference is large. For instance, for the 100K data set, the difference is 15 for $M = 3$. However, the difference lessens for a larger number of $M\ (\geqslant 6)$ and the index structure behavior is relative balanced. Fig. 6 shows the minimum, average and maximum heights for the 100K data set. According to this figure, for $M \geqslant 5$, these values are congruent and for $M \geqslant 9$ all are the equal implying the tree are balanced. It is worth noting that for $M \geqslant 5$, the difference between average and maximum heights is negligible and the tree is practically balanced. This is exactly what we expect based on the DMP-tree properties and the above discussion. When M is small, since no element can be in a higher level than its enclosure, the maximum tree height is expected to be high. With increasing M value, each node accommodates more data elements which can potentially be enclosures of each other. Therefore, the height of the tree is reduced and the tree becomes more compact.

Another performance measure that we computed was the memory utilization. Based on the node split policy, not all nodes are necessarily full and some of them may have only one data element. B-tree guarantees 50% memory utilization since it takes the median element as the split point. Thus, the data elements are evenly distributed between two resulted nodes. We cannot guarantee this for DMP-tree. However, Fig. 7 shows DMP-tree has good average memory utilization, from 64 to 68%. The average memory utilization is computed as the total data elements in the tree divided by the total possible number of data elements. Ultimately, it is worth noting that, as expected, the building processing of the tree is fast. On a heavily loaded SUN SPARC 10 it never took more 16 s to build the index tree.

### 6.2. *Results with general Alphabet*

Recently, we implemented DMP-tree with general alphabet set. The alphabet constituted of 256 possible characters in the ASCII system. We took about 40 text files from many sources like Internet RFCs, stories, etc. as the input. After eliminating redundant words, we had a database of 700K entries. We did the simulations using different 50K, 100K and 130K datasets.

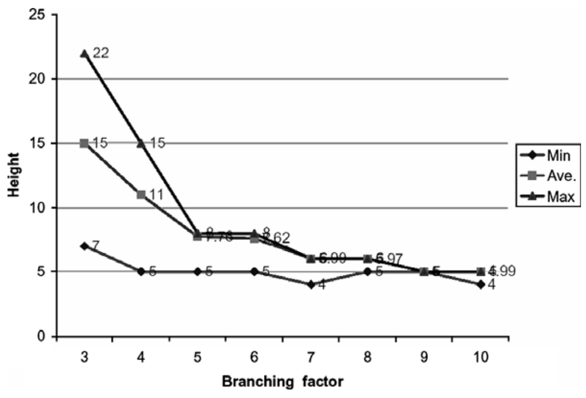

**Fig. 6.** The maximum, average and minimum heights of DMP-tree for 10^ {5} data prefixes of IP addresses.

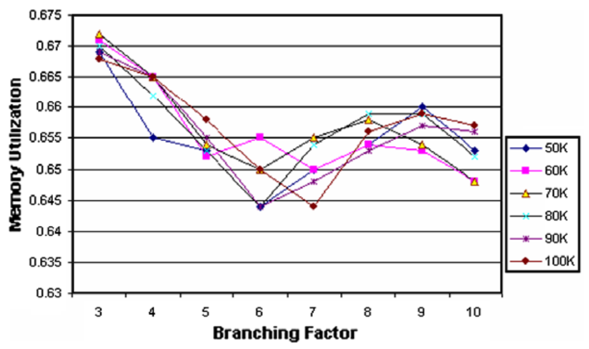

**Fig. 7.** The average memory utilization in the DMP-tree nodes for the data sample of IP address prefixes.

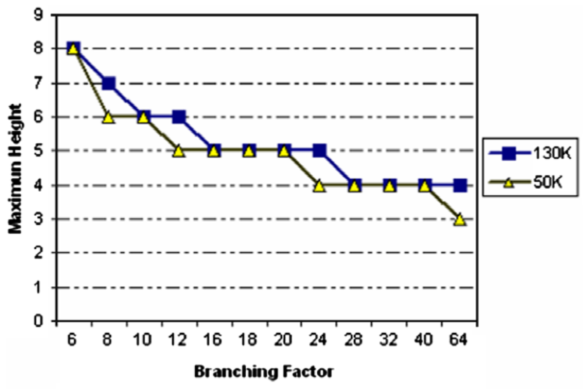

**Fig. 8.** maximum height of DMP-tree for 50K and 130K datasets.

Fig. 8, shows the maximum height of DMP-tree for 50K and 130K datasets with general alphabets.

Fig. 9, shows the average memory space utilization for different branching factors of DMP-tree with general alphabet. To get these results we have made DMP-tree for three datasets of 50K, 100K and 130K prefixes.

## 6.3. IP Lookup results using DMP-tree

We have designed and implemented some IP Lookup methods using DMP-tree. In [26] we have implemented DMP-tree-based IP Lookup in software and run it on a Pentium 4 an Alpha 21264B and an Athlon XP machine with Linux OS and gcc. The evaluation results got by *rdtscl* instruction showed that the method is shown in Table 2.

The results of Table 2 show that using a 2.5 GHZ Pentium 4, the lookup method can handle each packet in 320 ns. This means it can supports 6 Gbps lines assuming average packet size of 250 bytes. The result is very good for a software imple-

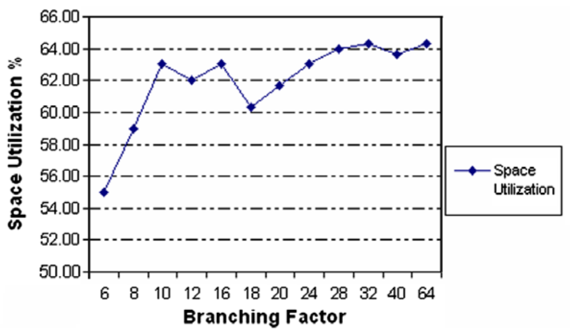

**Fig. 9.** Average memory space utilization for general alphabet.

**Table 2**
Average IP lookup time for a 100k routing table using DMP-tree

| Machine Name | Best result (Clock Cycles) |
|---|---|
| Alpha 21264B | 650 |
| Intel Pentium 4 | 800 |
| AMD Athlon XP | 700 |

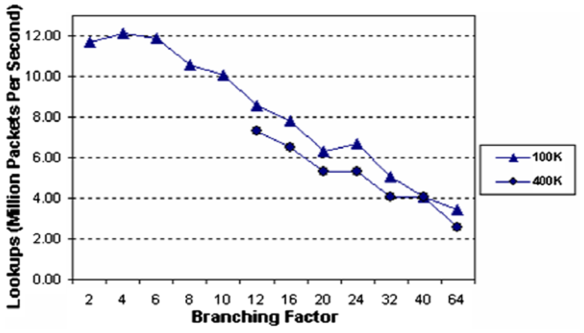

**Fig. 10.** HASIL Lookups in million packets per second for different braching factors of DMP-tree.

mentation and it will look more exciting when we notice the scalability of the method. In other word, if the routing table grows from 100K to 400K, only one level will be added to DMP-tree and it will involve just 15–20% of overhead.

Some hardware implementations of DMP-tree based IP lookup is done also. Ref. [30] presents a hardware architecture for an IP Lookup engine based on DMP-tree, It can perform each lookup in just one clock cycle. Thus, DMP-tree based IP lookup can be a very good, scalable, cheap and fast solution in hardware and it can be treated as a better alternative for TCAM-based methods.

Another approach for fast and scalable IP Lookup using DMP-tree, is HASIL [27], HASIL modifies memory unit of a general purpose processor to support two additional instructions. Fig. 10, shows forwarding capability of HASIL in million packets per second for 100K and 400K routing tables. It is worth noting that 1 million packets per second means about 2 Gbps.

The last approach was to modify functional units of a RISC processor to accelerate DMP-tree based lookup software [28,29]. It led to almost 30% improvement in lookup speed and 14% in code size.

## 7. Related work

Ref. [3] is a rich source for pattern and string matching problems and related algorithms. The general prefix string matching problem in which one is interested in finding the longest prefix of a pattern starting at each position of a text string is addressed in the pattern matching literature. This problem is a general case of the problems we address here and can be solved in linear time by adopting the string matching algorithm of Knuth, Morris and Pratt [15]. Ref. [7] studies the exact complexity and tight comparison bounds of this problem. Ref. [13] attacks this problem by proposing a combination of B-tree and Patricia tree as discussed previously in the paper. The method is mostly concerned with retrieving strings from the secondary memory. It starts with prefix matching and transfers it into the range query. It also transfers the general string matching problem into the prefix matching problem by taking suffixes of the data stings. Amir et al., [1], apply the same method to the dynamic dictionary matching. Ref. [5] store and uses prefixes to combine advantages of B-trees and digital search trees for the string search. The reader can refer to [6] which deals with the problem of matching sequences of different lengths and related problems. The main scheme for the prefix matching which is the base of some other methods and has been intensively discussed in the literature is Trie [16]. From the data structure point of view, a Trie is essentially an M-way tree where a branch in each node corresponds to a letter or character of alphabet $\Sigma$. The time and space complexity of the Trie structure is well known and has been discussed in [16]. The binary Trie [20] or the radix-tree [9] is the base of many methods for the IP lookup problem. The worst case search time in Trie is O(W) where W is the length of the longest string or IP address. A compacted Ttrie, [16] is built by removing internal nodes of a Trie that have a single child and concatenating the labels. Patricia Trie modifies the binary Trie by eliminating most of the unnecessary nodes [14]. The scheme has been implemented in the BSD kernel [21] and is the base of several IP lookup approaches proposed in the last few years. An interested reader can refer to [22,23,19,20,10,11,17] for more information.

## 8. Conclusion

We propose DMP-tree, Dynamic M-way Prefix tree, initially for the prefix matching. The prefix matching constitutes the base of some applications such as DNA matching, spell checking, online dictionaries, telephone directories and layers 3 and 4 switching. The well known IP lookup problem in TCP/IP packet forwarding has motivated us to propose the DMP-tree data structures. However, it has broader applications and may used in problems in which a set of strings from an alphabet $\Sigma$ can be prefixes of others as well as in general pattern matching. Previously proposed methods have used traditional sorting schemes for strings which are based on the character ordering in the alphabet and lengths of data strings. Unfortunately, those schemes cannot handle prefix matching directly and have to transfer them into range query which makes them very costly. The major problem preventing us from applying the usual indexing schemes such $B$ is the lack of a well known mechanism to sort strings of different lengths. We started our work by a new scheme for comparing and sorting strings of different lengths as a foundation. Then we build our index tree on top of the new sorting and comparison scheme.

DMP-tree is a superset of B-tree. When data elements are disjoint, meaning none of them is a prefix of another, it is the same as B-tree. The property which differentiates DMP-tree from B-tree is the node splitting policy. No data element can go to a higher level in node split if there is a string in the node which is its prefix. Therefore, no data element can be in a higher level than its prefix(es), called enclosure(s), in the index tree. The node splitting policy is restrictive in the sense that we cannot guarantee the final tree is balanced or it is possible to have 50% memory utilization in each node. However, our experimental result with some simulated binary data for the IP lookup application shows when the data set is large, over 50K, and the branching factor in the internal node is big (?5), DMP-tree acts like B-tree in terms of the search performance and memory utilization. Indeed, all branching factors that we ran gave good average memory utilization, 64–68%. However, the difference between the minimum and maximum height of the tree is large for small numbers of branching factors and decreases as the branching factor increases. Eventually, in some point, the tree is balanced. According to the experiments, the average height of DMP-tree is $O(\text{Log}\,\{M, N)$ for large $N$ and big branching factor M in the internal nodes.

DMP-tree is applicable to a wide range of applications and handles a large range of queries. The data structure is very efficient for time critical applications such as IP lookup where the longest prefix matching a query string must be found in a short time. It also efficiently finds the shortest prefix or all prefixes matching a given query string in a time proportional to the height of the tree, $O(\text{Log}\,\{M, N)$. DMP-tree can also be used to search all strings having a special prefixes. This query is the base of the general pattern matching algorithm. We showed how the scheme can be extended and applied to pattern matching in an efficient way.

As previously discussed, in the best case, DMP-tree is like B-tree with average height of $O(\text{Log}\,\{M, N)$ and minimum node utilization of 50%. We do not have any analytical results for the worst and average case of the tree for the search, or height of the tree or node utilization, even though our experimental results shows it is like B-tree for large number of data and big branching factor in the internal node.

## References

[1] Amir A, Farach M, Galil Z, Gaincarlo R, Park K. Dynamic Dictionary Matching. J Comput Syst Sci 1994;49:111–5.
[2] Alleva, Fileno A. Method and system for encoding pronunciation prefix trees, US Patent 5,758,024, June 25; 1996.
[3] Apostolico Alberto, Galil Zvi. Pattern matching algorithms. Oxford University Press; 1997.
[4] Bayer R, McCreight C. Organization and maintenance of large ordered indexes. Acta Inform 1972;1(3):173–89.
[5] Bayer R, Unterauer K. Prefix B-tree. ACM Trans Database Sys 1977;2(1):11–26.
[6] Bozkaya Tolga, Yazdani Nasser, Ozsoyoglu Z Meral. Sequence matching of different lengths. In: 6th International conference on information and knowledge management (CIKM'97), November; 1997.
[7] Breslauer D, Colussi L Toniolo L. Tight comparison bounds for the string prefix matching problem. In: Proceeding of combinatorial pattern matching, 4th symposium; 1993.
[8] Carroll MartinD, Juhl Peter, Koenig AndrewRichard. Method and apparatus for parsing source code using prefix analysis, US Patent 5,812,853; April 11, 1994.
[9] Cormen Thomas H, Leiserson Charles E, Riverst Ronald L. Introduction to algorithms. The MIT press; 1990.
[10] Degermark Mikael, Brondnik Andrej, Carlson Svante, Pink Stephen. Small forwarding tables for fast routing lookups. In: Proceeding of SIGCOMM; 1997.
[11] Doeringer W, Karjoth G, Nassehi M. Routing on longest-matching prefixes. IEEE/ACM Trans Network 1996;4(1):86–97.
[12] Eastman WillardL, Lempel Abraham, Ziv Jacob, Cohn Martin. Apparatus and method for compressing data signals and restoring the compressed data signals, US Patent 4,464,650; August 10, 1981.
[13] Ferragina Paolo, Grossi Roberto. The string B-tree: a new data structure for string search in external memory and its applications. J ACM 1999;46(2).
[14] Gonnet GH, Baeza-Yates RA. Handbook of algorithms and data structures. second ed. Addison Wesley; 1991.
[15] Knuth DE, Morris JH, Pratt VR. Fast pattern matching in strings. SIAM J Comput 1977;6:322–50.
[16] Knuth Donald E. The art of programming. Sort Search, vol. 3. Addison Wesley; 1973.
[17] Lampson B, Srinivasan V, Varghese G. IP lookups using multiway and multicolumn search 1998.
[18] Manber U, Myers G. SuÆx arrays: a new method for on-line string searches. SIAM J Comput 1993;22(5):935–48.
[19] Nilsson S, Karlsson G. Fast address look-up for Internet routers. In: Proceedings of IEEE broadband communication 98; April 1998.
[20] Nilsson S, Karlsson G. Implementing a dynamic compressed Trie. In: Proceedings of WAE'98, Saarbrucken, Germany; August 1998.
[21] Sklower K. A tree-based routing table for Berkeley Unix. In: Proceeding of the winter Usenix conference; 1991.
[22] Srinivasan V, Varghese George. Fast address lookups using controlled prefix. In: Proceedings of ACM sigmetrics; September 1998.
[23] Waldvogel Marcel, Varghese George, Turner Jon, Plattner Bernhard. Scalable high speed IP routing lookups. In: Proceedings of ACM sigcomm; September 1997.
[24] Wilkinson, III Hugh M, Varghese George, Poole Nigel T. Compressed prefix matching database searching, US Patent 5,781,772; May 15, 1995.
[25] Yazdani Nasser, Min Paul S. Prefix trees: new efficient data structures for matching strings of diffierent lengths. In: Proceedings of the IEEE database engineering conference (IDEAS 01). Grenoble, France; July 2001.
[26] Yazdani Nasser, Mohammadi Hossein. IP lookup in software for large routing tables using DMP-tree data structure. In: Proceeding of 9th IEEE-APCC 2003, Penang, Malaysia; 2003.
[27] Mohammadi Hossein, Yazdani Nasser, Robatmili Behnam, Nourani Mehrdad. HASIL: hardware assisted software-based IP lookup for large routing tables. Proceeding of the 11th IEEE international conference on networks (ICON) 2003, Sydney, Australia. p. 99–105.

[28] Ghasemi Hamid Reza, Mohammadi Hossein, Robatmili Behnam, Yazdani Nasser. Augmenting general purpose processors for network processing. In: Proceeding of the second IEEE international conference on field-programmable technology (FPT 2003), Tokyo, Japan; 2003.
[29] Mohammadi Hossein, Robatmili Bahnam, Ghasemi Hamid Reza, Yazdani Nasser, Nourani Mehrdad. Line speed IP lookup in software using improved functional units. In: Proceeding of the 9th CSICC 2004, Tehran, Iran. p. 104–10.
[30] Yazdani Nasser, Salimi Nazila. Performing IP lookup on very high line speed. In: Proceeding of ICT 2002, Shiraz, Iran; 2002.
[31] Yazdani Nasser, Min Paul S. Fast and salable schemes for IP lookup problem. In: Proceedings of the IEEE conference high performance switching and routing, Heidelberg Germany; 2000.