Journal of Computer and Communications, 2016, 4, 134-145
Published Online April 2016 in SciRes. http://www.scirp.org/journal/jcc
http://dx.doi.org/10.4236/jcc.2016.44012

Prime Box Parallel Search Algorithm: Searching Dynamic Dictionary in O(lg m) Time

Amit Pandey, Berhane Wolde-Gabriel, Elias Jarso

School of Informatics, IOT, Hawassa University, Hawassa, Ethiopia
Email: amit.pandey@live.com

Received 13 March 2016; accepted 24 April 2016; published 27 April 2016

Abstract

Hashing and Trie tree data structures are among the preeminent data mining techniques considered for the ideal search. Hashing techniques have the amortized time complexity of O(1). Although in worst case, searching a hash table can take as much as θ(n) time [1]. On the other hand, Trie tree data structure is also well renowned data structure. The ideal lookup time for searching a string of length m in database of n strings using Trie data structure is O(m) [2]. In the present study, we have proposed a novel Prime Box parallel search algorithm for searching a string of length m in a dictionary of dynamically increasing size, with a worst case search time complexity of O($\log_2$m). We have exploited parallel techniques over this novel algorithm to achieve this search time complexity. Also this prime Box search is independent of the total words present in the dictionary, which makes it more suitable for dynamic dictionaries with increasing size.

Keywords

Prime Box Search Algorithm, Information Retrieval, Lexicographical Search, Dynamic Dictionary Search, Parallel Search

1. Introduction

A static dictionary is a subset of a finite universe with fixed set size, that is the static dictionary can be considered as collection of words with bounded size. This limit that bounds the size can be huge. In existing approaches, the lexicographical search techniques are used for word look up in dictionaries, that is why the search time complexity in case of static dictionary is directly proportional to its size.

Now consider a dictionary with huge word set and chronically increasing size, that is the size of the word set in the dictionary is not fixed. Consider the example of English dictionary that has more than 600,000 words

How to cite this paper: Pandey, A., Wolde-Gabriel, B. and Jarso, E. (2016) Prime Box Parallel Search Algorithm: Searching Dynamic Dictionary in O(lg m) Time. *Journal of Computer and Communications*, **4**, 134-145.
http://dx.doi.org/10.4236/jcc.2016.44012

inherited from the language and keeps on increasing. A new word lookup approach that is free from total number of words is required for such dynamic dictionaries. In present scenario we have many approaches based on trie tree data structure and hashing techniques for word lookup in static dictionary. But when it comes to search a dynamic dictionary with continuously increasing size, there is not any satisfactory technique yet with promising results.

In the present study we have proposed parallel prime box search algorithm that will use a unique combination of prime numbers assigned to each word to identify it. Further we have shown that the complexity of this proposed parallel algorithm is $O(\log_2 m)$. The proposed technique depends on the length of the word and is independent of the size of the dictionary and therefore best suits the needs of dynamic dictionaries.

2. Related Work

An adequate amount of work has been done on the techniques for information retrieval and dictionary search. Researchers are working in this field for more than last fifty years. E. Fredkin [3] and D. E. Knuth [4] has proposed early techniques to read the input character by character. In 1987 J. Aoe and M. Fujikawa proposed how to lookup up a word in a morphological dictionaries [5]. Further J. L. Peterson proposed a spelling checker [6], and J. Aoe, Y. Yamamoto and R. Shimada proposed an approach in mid 80's for reading character by character input for lexical analysis in compiler or a bibliographic search [7] [8]. In 2009 Georgios Stefanakis also proposed an implementation of range trie for address lookups [9]. Most of the above work was either based on or was related to Trie structure. In Trie structure each node of the trie tree is an array, such that one character from each string is stored in an array at each level. A trie tree is presented in **Figure 1** for the strings "ace", "fade", "face" and "fact". Here each node is represented by an array at different levels [10] [11]. To search a string of length m in a trie tree, one has to make m comparisons. One comparison at each level, making its search time as O(m).

The single array representation of trie datastructure is expensive in sense of storage complexity. As storage space is directly proportional to the number of nodes. So to overcome this problem a list structure was proposed for trie tree [12]. This list structure was implemented using the linked lists for better space usage. Further a double array structure for trie tree was proposed [13]. In which two arrays BASE and Check were used to keep records of nodes to provide a compact as well as fast access.

Douglas Comer [14] has also proposed hashing based technique for faster dictionary search. In 2001 Rasmus Pagh [15] has proposed minimal perfect hash function with constant query time. Later in 2009 Djamal Belazzougui *et al*. [16] has also proposed minimal perfect hashing based approaches for searching in a sorted table in O(1) query time. A minimal perfect hash function is a bijective function such that if ω is a set of n words {w1, w2, w3, ---, wn}. Then the minimal perfect hash function α will map each word of the set ω uniquely in to set having index values {0, 1, 2, 3, ---, (n − 1)} [17] [18]. But all this work was done for static dictionaries, which are subset of a finite universe with fixed set size.

Amihood Amir *et al.* in 1994 has proposed a dynamic dictionary matching technique with search time complexity of $O((n + tocc) \log |D_i|)$. Where *tocc* is total occurrences of the word in the text [19]. In 2014, Chouvalit Khancome and V. Boonjing have also proposed an inverted list based dynamic dictionary searching algorithm for the lookup in linear time [20].

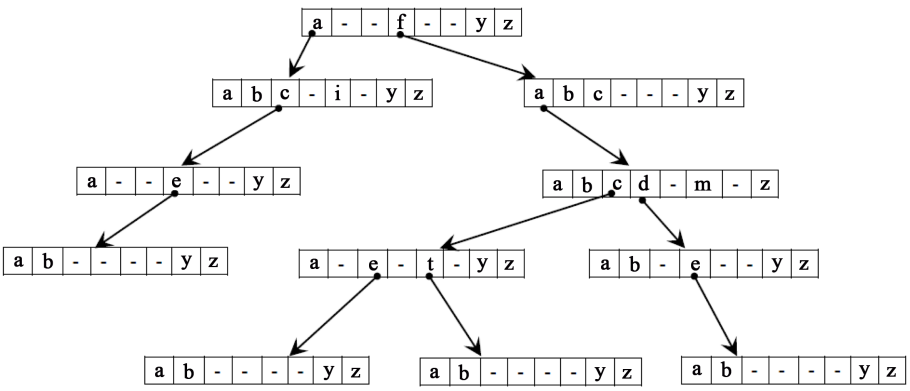

Figure 1. An example of a trie tree.

In this paper we have proposed a novel approach for searching dynamic dictionaries in $O(\log_2 m)$ time. The new approach is independent of the size of the dictionary and therefore well suites the dynamic dictionary search

3. Proposed Approach

Here in prime box algorithm we will use a unique number assigned as key to each word, which is further used as array index to find that word. This approach of lookup puts the prime box algorithm in a category of directly addressed array lookup. Consider a scenario where you are looking for a word in a dictionary. Now to assign a unique key to that searched word we have to take help of prime box. This is an "m × 26" table of prime numbers, where each prime number appears only once. Here m is the length of longest word present in the dictionary or the database. Let us assume the largest word present in the dictionary is of length three characters. Then our prime box will look like as presented in **Table 1**.

To calculate a unique key for a word the prime numbers associated with each character at different levels from the prime box table are multiplied. To calculate the key for the word "AIM". The prime number associated with "A" at level 1, prime number associated with "I" at level 2 and the prime number associated with "M" at level 3 in the prime box table will be multiplied together. The prime number for "A" in the table is "2", for "I" it is "149" and for "M" the value is "313". So the unique key for the word "AIM" can be calculated as "2 × 149 × 313", which is equal to "93274". As these keys only have prime factors, this ensures that the key values will be unique for each word. Further using these key values as the array indices the associated words will be saved at particular locations in array to create a word data structure. Then the direct addressing lookup scheme will be used to find out the word using the same key value.

Proposed Parallel Prime Box Algorithm: Searching in $O(\log_2 m)$ Time

In multiprocessor environment we can easily employ the parallel computing techniques over the independent for loops of the proposed algorithm. The FOR loops below do not have any dependent instruction or variable between them. This makes it possible to easily apply the parallel techniques over them to enhance the performance.

```
M: The length of the longest word present in the language set.
x: ASCII value for the first alphabet of the language set.
y: ASCII value for last alphabet of the language set.
Unique_Prime( ) = Function returning a Unique Prime Number Each Time.
Prime_Box: Two dimensional m x 26 array of unique prime numbers.
Word_Struct[n] = Array of length n, Initialized all to 1.
Unique_Prime_Index = Unique Prime number used as word's address in the array. Initialized to 1.
Location[ ] = Array storing the words of dynamic dictionary.

[Loop 1]For □i : i = 0 to i < M do in Parallel
                    For □j : j = x to j = y do in Parallel
                        Prime_Box[i][ j- x] = Unique_Prime( )

PART 1: Building the Data Structure

[Loop 2]For □j : j = 0 to j < Input_Word.length do in Parallel
U = Prime_Box[ j][(ASCII Value of j^th alphabet) - x]
        Unique_Prime_Index = Unique_Prime_Index * U

        Location[Unique_Prime_Index] = 1
        Set Unique_Prime_Index = 1

PART 2: The Search Algorithm

[Loop 3]    For □j : j = 0 to j < Input_Word. length do in Parallel
U = Prime_Box[ j][(ASCII Value of j^th alphabet) - x]
        Unique_Prime_Index = Unique_Prime_Index * U

If ( Location[Unique_Prime_Index] = = 1)
Then, return ( True )
else return ( False )
Set Unique_Prime_Index = 1

PART 3: The Deletion Algorithm

[Loop 4]    For □j : j = 0 to j <   Input_Word.length   do in Parallel
U = Prime_Box[ j][(ASCII Value of j^th alphabet) - x]
        Unique_Prime_Index = Unique_Prime_Index * U

        Location[Unique_Prime_Index] = 0
        Set Unique_Prime_Index = 1
```

In Loop 1, Loop 2, Loop 3 and Loop 4, it can be seen clearly that the FOR loop will be working in parallel for calculating the Unique_Prime_Index value. Actually for all the alphabets present in the given word of size 'm', distinct “m” prime numbers will be picked from the prime box and multiplied together in parallel to obtain the Unique_Prime_Index value. This Unique _Prime_Index value will be used further as array index for fetching the value at that location. If that value is one then the word is present otherwise it is absent.

4. Complexity of the Algorithm

As the above parallel algorithm advances. The index value for the word to be searched will be calculated by available processors in steps as shown below in **Figure 2**. Here the proposed algorithm is implemented using the conceptual balanced binary tree designing technique, usually recognized as recursion tree [21].

As the conceptual balanced binary tree model is followed, the total number of nodes in the tree can be expressed as,

$$n = \left(2^h + 1\right) - 1 \tag{1}$$

here, h is the height of the tree.

Also the total number of leaf nodes in a balanced binary tree can be expressed as,

$$L = 2^h \tag{2}$$

where, h is the height of the tree.

As there will be m prime numbers to be multiplied, each belonging to an alphabet in the word. And multiplication is a binary associative operation. So there will be $(m/2)$ nodes in the tree at the leaf level.

$$\text{So,} \quad L = \left(m/2\right) \tag{3}$$

Now using expression (2) and (3) the expression (1) can be modified and rewritten as,

$$n = 2^h + 2^h - 1$$

$$\text{or,} \quad n = 2 \cdot L - 1$$

$$\text{or,} \quad n = 2 \cdot \left(m/2\right) - 1$$

$$\text{or,} \quad n = m - 1 \tag{4}$$

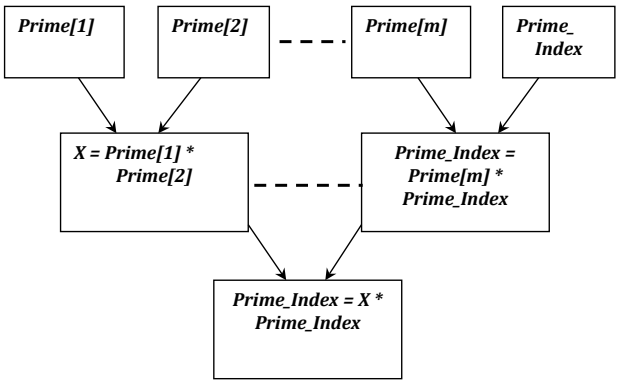

Figure 2. Diagram showing parallel computation process for array index.

Table 1. Prime Box for maximum word length of three characters.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	-	-	X	Y	Z
1	2	3	5	7	11	13	17	19	23	29	31	37	41	43	-	-	89	97	101
2	103	107	109	113	127	131	137	139	149	151	157	163	167	173	-	-	229	233	239
3	241	251	257	263	269	271	277	281	283	293	307	311	313	317	-	-	283	389	397

Using expression (1) and (4), We conclude that,

$$\left(2^{h+1}\right)-1=m-1$$

$$\text{or,}\quad \left(2^{h+1}\right)=m \tag{5}$$

Now taking log of both sides of the expression (5),

$$\log_2\left(2^{h+1}\right)=\log_2\left(m\right)$$

$$\text{or,}\quad \left(h+1\right)\cdot\log_2\left(2\right)=\log_2\left(m\right)$$

$$\text{or,}\quad h+1=\log_2\left(m\right)$$

$$\text{or,}\quad h=\log_2\left(m\right)-1 \tag{6}$$

Here, the height of the tree represents the total time needed for performing calculation. As height of the tree represents the total number of steps required for calculation and each step is performed in a unit time. Hence we can say that the time complexity for loop 3 in part 2 of the proposed algorithm is O($\log_2 m$) using O(m) processors. Here m is length of the input word. Also it is easy to see that the remaining steps in part 2 will take unit time. Hence making the overall search time complexity of the algorithm in part 2 as O($\log_2 m$).

In case of loop 1, the operation performed in the inner loop will only take unit time. Which will make the overall time complexity of the loop 1 as O(1) using O(MV) processors [22]. Here M is length of the longest word present in the language set and V is total number of alphabets in the language set.

Like loop 3, loop 2 in part 1 of the algorithm will also take O($\log_2$m) time using O(m) processors [22]. Now considering the time complexity of the loop 2 in part 1 and acknowledging the fact that other operations that are outside the loop 2 in part 1 will take only unit time. We can state that the overall time complexity for building the datastructure in part 1 is also O($\log_2 m$).

Now, for the deletion algorithm in part 3. The loop 4 will have the time complexity of O($\log_2 m$) using O(m) processors, as we have shown above for similar cases. The remaining steps of the part 3 will be executed in O(1) time. Hence for this part also the overall time complexity will be O($\log_2 m$).

Hence, we can state that the proposed Prime Box algorithm can be solved in O($\log_2 m$) time using at least O(m/$\log_2 m$) processors [22]. Now it is also clear that by using this parallel approach we have improved the time complexity of the algorithm to O($\log_2 m$). Which in the case of sequential approach would have been O(m).

5. Proving Correctness of the Search Algorithm

We will use mathematical Induction to prove the correctness of this search algorithm. We have to start from loop 3 in part 2 of the proposed algorithm. First we have to prove the precondition, that is searching a word of unit length. When the length of the input word is one. The for loop will run for only one time and then terminate. In the run only one step will be evaluated in unit time. In which two operands the Unique_Prime_Index with value one and the prime number from the prime box will be multiplied with each other to get the unique array index for the word. Hence the overall time taken will be of the order O(1) for the loop 3 in this case. Also each step in the algorithm that are outside loop 3 will be evaluated in unit time. Hence we can state that we can search a word of unit length in O(1) time. This can again be verified, as we know that the overall search time complexity is O($\log_2 m$). The value of operands here are two. So the overall time complexity for this precondition will be O($\log_2 2$) or O(1).

Now we will prove that the above search algorithm will also work for some word of length (K + 1). As the word has length of (K + 1). So the number of leaf nodes in the conceptual balanced binary tree will be (K + 1)/2, because multiplication is binary associative operation. Now using expression (2) and (3) the expression (1) can be modified and rewritten as,

$$n=2^h+2^h-1$$

$$\text{or,}\quad n=2\cdot L-1$$

$$\text{or,}\quad n=2\cdot\left(\left(K+1\right)/2\right)-1$$

$$\text{or,} \quad n = K \tag{7}$$

Using expression (1) and (7), We conclude that,

$$\left(2^{h+1}\right) - 1 = K$$

$$\text{or,} \quad \left(2^{h+1}\right) = K + 1 \tag{8}$$

Now taking log of both sides of the expression (8),

$$\log_2\left(2^{h+1}\right) = \log_2\left(K+1\right)$$

$$\text{or,} \quad \left(h+1\right)\cdot\log_2\left(2\right) = \log_2\left(K+1\right)$$

$$\text{or,} \quad h+1 = \log_2\left(K+1\right)$$

$$\text{or,} \quad h = \log_2\left(K+1\right) - 1 \tag{9}$$

The for loop in loop 3 of part 2 will finally terminate after iterating for (K + 1) times, as (K + 1) is the length of the given input. Now, concluding from expression (9), we can say that time complexity in this case is $\log_2(K + 1)$. We also know that here length of the word m = (K + 1). So again we have proven that the time complexity for the loop 3 in part 2 is $O(\log_2 m)$. Further it is easy to see that the remaining steps of the search algorithm will be finished in O(1) time. Resulting in the overall time complexity of the search algorithm as $O(\log_2 m)$. So we can see that a word of (K + 1) length can be searched in $\log_2(m)$ time.

Hence by mathematical induction it is proved that for any given word of length m, we can search it in a time of $O(\log_2 m)$.

6. Applying Parallel Prime Box Algorithm

The Proposed algorithm has three parts. First is building the data structure, where insertion of words takes place, second is the search algorithm and third part is associated with the deletion of word from the datastructure. We can also update any word from the data structure if required. Actually this datastructure updating requires two functions working consecutively. First doing the deletion and then inserting the new word. We will see the working of the algorithm step by step starting with the insertion of the word "an".

Before we start first in loop 1 the Prime box data structure is created that is a Two dimensional m *x* 26 array of unique prime numbers stored in it.

6.1. Building the Datastructure: Inserting the Word "an"

Now we enter loop 2. Here a prime number for each alphabet of the word is retrieved from the prime box datastructure and multiplied together to obtain the Unique_Prime_Index value. Then at that index in Location array we mark its entry true.

Here, using the prime box datastructure mentioned in **Table 1**. We will get the prime values for "a" as 2 and for "n" as 173. Then these values will be multiplied to get a Unique_Prime_Index value for the word "an" as 346. Then we set its entry true by marking Location [346] = 1 (See **Figure 3**).

6.2. Search Algorithm: Searching the Word "an"

We enter loop 3. Here a prime number for each alphabet of the word is retrieved from the prime box datastructure and multiplied together to obtain the Unique_Prime_Index value. Then we check at that index in Location array whether it is true or not. If it is true then we return that word is present otherwise we return false.

Here, using the prime box datastructure mentioned in **Table 1**. We will get the prime values for "a" as 2 and for "n" as 173. Then these values will be multiplied to get a Unique_Prime_Index value for the word "an" as 346. Then we check the entry at Location [346]. If it is "1" then return that the word is present otherwise return false (See **Figure 4**).

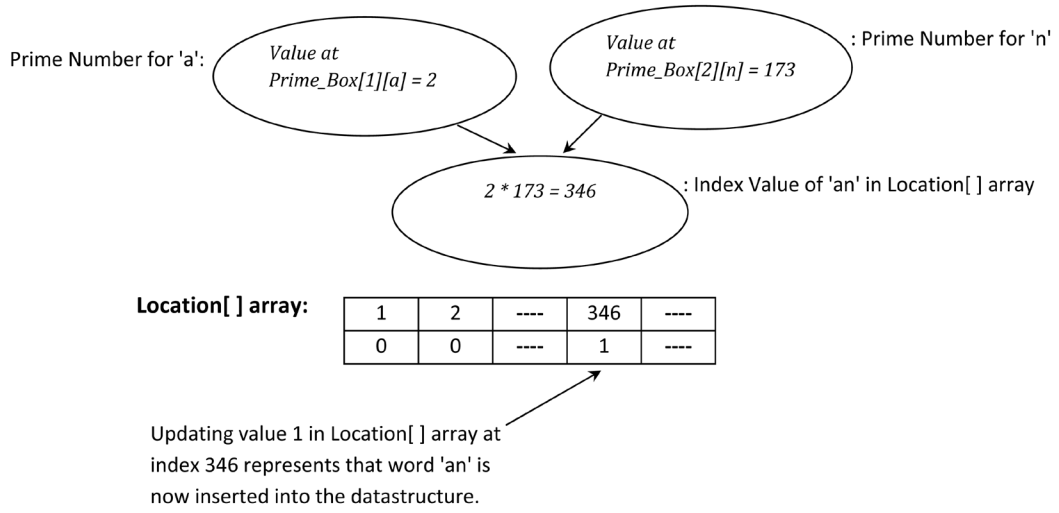

Figure 3. Building the datastructure: Inserting the word “an”.

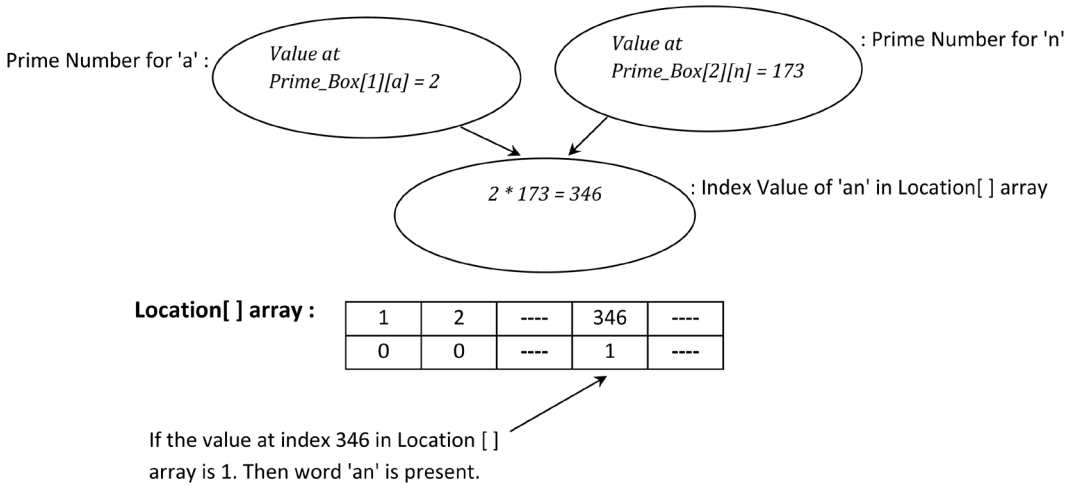

Figure 4. Search Algorithm: Searching the word “an”.

6.3. Deletion Algorithm: Deleting the Word “an”

We enter loop 4. Here a prime number for each alphabet of the word is retrieved from the prime box datastructure and multiplied together to obtain the Unique_Prime_Index value. Then at that index in Location array we mark its entry False.

Here, using the prime box datastructure mentioned in **Table 1**. We will get the prime values for “a” as 2 and for “n” as 173. Then these values will be multiplied to get a Unique_Prime_Index value for the word “an” as 346. Then we set its entry false by marking Location [346] = 0 (See **Figure 5**).

7. Results

We have implemented the parallel approaches using C# and executed the code to get the data for different words with varying word length from the English dictionary. The column 'word' has the searched words and another column 'search time' has the respective listing of the time taken by the algorithm to search those words. Here we have not mentioned any fixed database size or total number of words in the dictionary, as our novel approach is independent of the total size.

With our current implementation in C#. It is only possible to lookup a word till word size of four. As increasing the size further will create the index values that are outside the current range of the arrays defined in C#.

We can see the data obtained after executing the code for Prime Box algorithm in **Table 2** (See **Table 2**). The plot for the search time is logarithmic graph, with another curved line for C*$\log_2$m as its upper bound (See **Figure 6**). This confirms that the time complexity for the proposed parallel algorithm is O($\log_2$m).

Eventually after analyzing the graph for the insertion time (See **Figure 7**) and deletion time (See **Figure 8**) of the algorithm. It is clear that the graph for C*$\log_2$(m) is the upper bound over both the graphs of insertion time and deletion time. This justifies their O($\log_2$m) time complexity.

Further we have implemented the Trie search algorithm in C#. The search time values for Trie algorithm together with the search time values for Prime Box algorithm are presented in **Table 3** (See **Table 3**). We have considered Trie algorithm for comparison with the proposed algorithm as Trie algorithm is also free from the total number of words present in the dictionary and hence is also suitable for dynamic dictionary search. In the plots for both the algorithms (See **Figure 9**). The Trie algorithm has a linear graph with positive slope justifying its O(m) time complexity and Prime Box algorithm has a logarithmic curve justifying its O($\log_2$m) time complexity. It is clearly visible from the graphs in **Figure 9** (See **Figure 9**) that the proposed Prime Box algorithm is more efficient than Trie search algorithm because of its smaller search time.

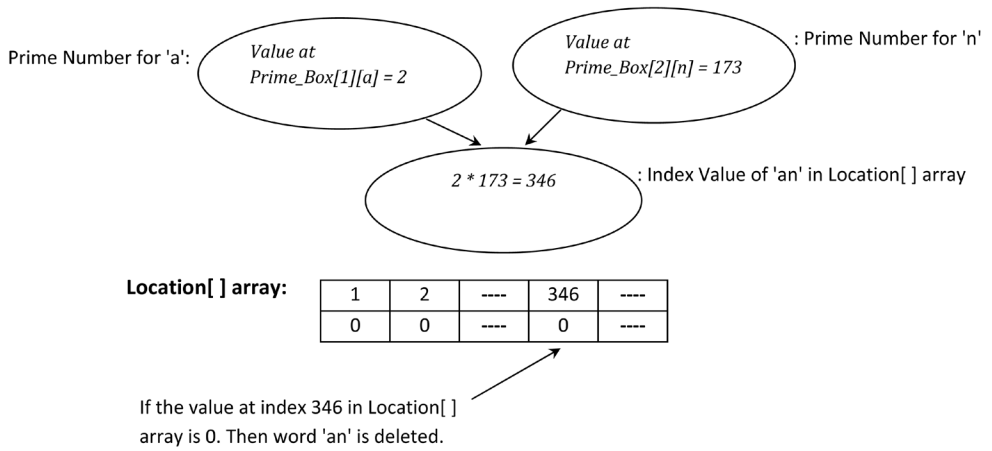

Figure 5. Deletion Algorithm: Deleting the word “an”.

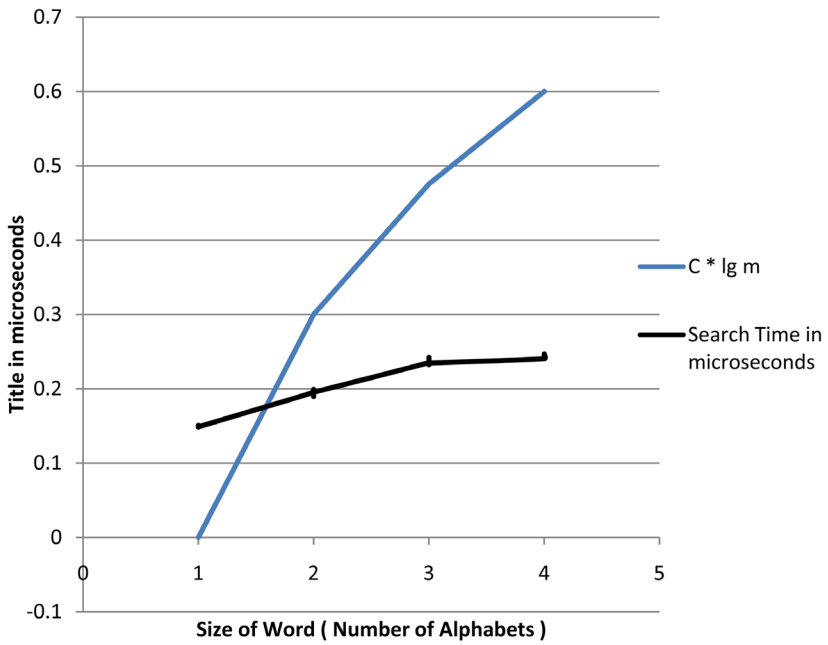

Figure 6. Graph for search time of prime box Algorithm: Search time is O(lg m).

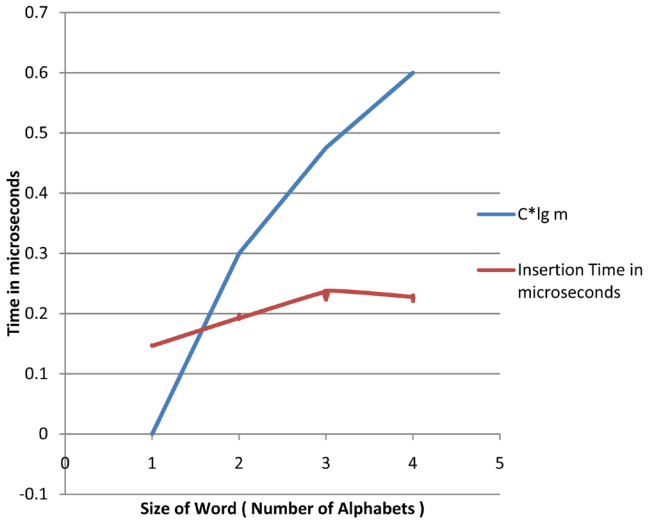

Figure 7. Graph for insertion time of prime box Algorithm: Insertion time is O (lg m).

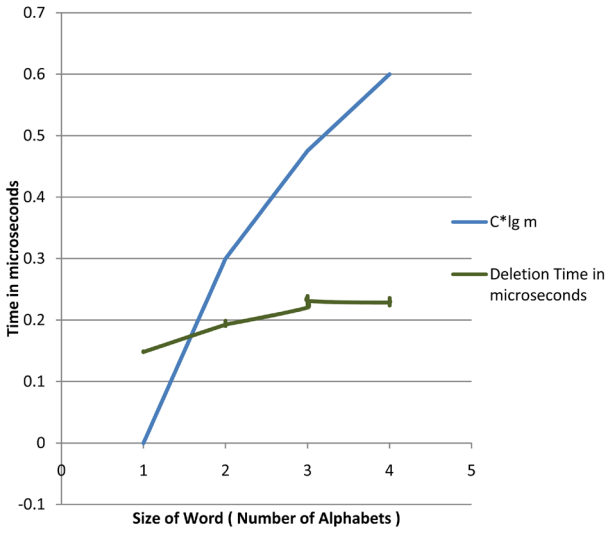

Figure 8. Graph for deletion time of prime box Algorithm: Deletion time is O (lg m).

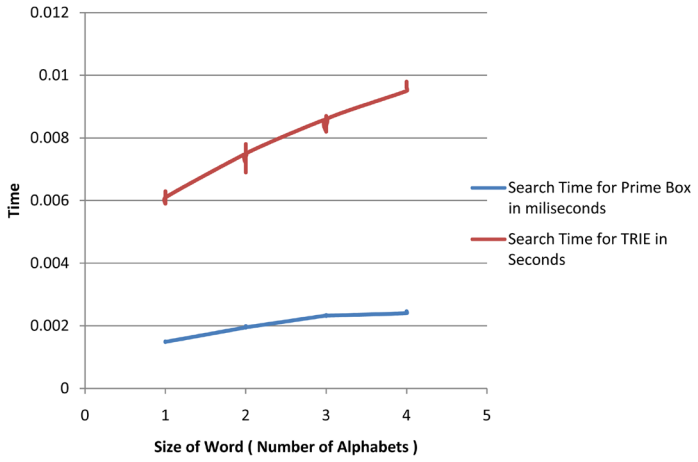

Figure 9. Comparing search time graphs for TRIE and prime box parallel search Algorithm.

Table 2. Search times for both sequential and parallel approaches.

Insertion Using Prime Box Algorithm			Search Using Prime Box Algorithm			Deletion Using Prime Box Algorithm		
Word	Search Time (microseconds)	C*lg(m) For C = 0.3	Word	Search Time (microseconds)	C*lg(m) For C = 0.3	Word	Search Time (microseconds)	C*lg(m) For C = 0.3
a	0.1477	0	a	0.1505	0	a	0.1476	0
i	0.1468	0	i	0.1481	0	i	0.1491	0
o	0.1465	0	o	0.1487	0	o	0.1481	0
an	0.1928	0.3	an	0.195	0.3	an	0.1924	0.3
as	0.1911	0.3	as	0.1971	0.3	as	0.1902	0.3
at	0.1915	0.3	at	0.1967	0.3	at	0.1988	0.3
be	0.1962	0.3	be	0.1962	0.3	be	0.1915	0.3
by	0.1984	0.3	by	0.195	0.3	by	0.1971	0.3
do	0.1954	0.3	do	0.1941	0.3	do	0.1947	0.3
go	0.1928	0.3	go	0.1984	0.3	go	0.1963	0.3
he	0.1962	0.3	he	0.1958	0.3	he	0.1972	0.3
if	0.1941	0.3	if	0.1967	0.3	if	0.1935	0.3
in	0.1962	0.3	in	0.1984	0.3	in	0.1954	0.3
is	0.1924	0.3	is	0.1979	0.3	is	0.1928	0.3
you	0.2364	0.475489	you	0.2343	0.475489	you	0.2202	0.475489
all	0.2245	0.475489	all	0.242	0.475489	all	0.2322	0.475489
any	0.2351	0.475489	any	0.237	0.475489	any	0.2364	0.475489
can	0.2351	0.475489	can	0.2356	0.475489	can	0.221	0.475489
her	0.2347	0.475489	her	0.2344	0.475489	her	0.2279	0.475489
was	0.2262	0.475489	was	0.2357	0.475489	was	0.2266	0.475489
one	0.2283	0.475489	one	0.2337	0.475489	one	0.2364	0.475489
our	0.2236	0.475489	our	0.2332	0.475489	our	0.239	0.475489
get	0.2381	0.475489	get	0.2345	0.475489	get	0.2309	0.475489
army	0.2275	0.6	army	0.2465	0.6	army	0.2283	0.6
atom	0.224	0.6	atom	0.2457	0.6	atom	0.233	0.6
aunt	0.2249	0.6	aunt	0.2445	0.6	aunt	0.236	0.6
aura	0.2283	0.6	aura	0.2465	0.6	aura	0.2347	0.6
auto	0.221	0.6	auto	0.2442	0.6	auto	0.2253	0.6
bait	0.231	0.6	bait	0.2452	0.6	bait	0.2334	0.6
bank	0.2266	0.6	bank	0.2449	0.6	bank	0.2236	0.6

Table 3. Comparing search time of TRIE with prime box algorithm.

TRIE Search		Prime Box Parallel Search	
Word	Search Time (Seconds)	Word	Search Time (miliseconds)
a	0.0063	a	0.001505
i	0.0059	i	0.001481
o	0.0061	o	0.001487
an	0.0075	an	0.00195
as	0.0074	as	0.001971
at	0.0073	at	0.001967
be	0.0074	be	0.001962
by	0.0074	by	0.00195
do	0.0076	do	0.001941
go	0.0072	go	0.001984
he	0.0074	he	0.001958
if	0.0073	if	0.001967
in	0.007	in	0.001984
is	0.0069	is	0.001979
it	0.0071	it	0.001994
me	0.0073	me	0.001992
my	0.0074	my	0.001945
or	0.0075	or	0.001958
ox	0.0074	ox	0.001945
so	0.0078	so	0.001971
to	0.0076	to	0.001988
up	0.0072	up	0.001992
we	0.0075	we	0.001954
the	0.0086	the	0.002329
and	0.0084	and	0.002323
for	0.0085	for	0.002324
are	0.0084	are	0.002323
but	0.0083	but	0.002326
not	0.0086	not	0.002342
you	0.0085	you	0.002321
all	0.0087	all	0.002313
any	0.0084	any	0.002345
can	0.0086	can	0.002336
her	0.0085	her	0.002324
was	0.0083	was	0.002347
one	0.0084	one	0.002337
our	0.0082	our	0.002342
get	0.0086	get	0.002325
abed	0.0095	abed	0.002404
acme	0.0096	acme	0.002462
agar	0.0098	agar	0.002416
also	0.0096	also	0.002469
area	0.0097	area	0.002404
army	0.0098	army	0.002465
atom	0.0098	atom	0.002457
aunt	0.0097	aunt	0.002445
aura	0.0098	aura	0.002465
auto	0.0097	auto	0.002442
bait	0.0096	bait	0.002452
bank	0.0097	bank	0.002449

8. Conclusions

Analyzing data and graphs provided, it is inferred that it is possible to lookup a word in a dynamic dictionary in $O(\log_2 m)$ time using the prime box parallel search algorithm. Unlike any other algorithm, this approach is independent of the total number of words present in the dictionary and hence well suits the need for searching a dynamic dictionary with increasing size.

Our main emphasis while designing this algorithm was to minimize the time complexity. Hence as future work, still there is scope for further improvement in the algorithm in terms of its space complexity.

References

[1] Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C. (2001) Introduction to Algorithms. 2nd Edition, The MIT Press, London.

[2] Maass, M.G. (2006) Average-Case Analysis of Approximate Trie Search. *Algorithmica*, **46**, 469-491. http://dx.doi.org/10.1007/s00453-006-0126-4

[3] Fredkin, E. (1960) Trie Memory. *Communications of the ACM*, **3**, 490-499. http://dx.doi.org/10.1145/367390.367400

[4] Knuth, D.E. (1973) The Art of Computer Programming, Vol. III: Sorting and Searching.

[5] Aoe, J.I. and Fujikawa, M. (1987) An Efficient Representation of Hierarchical Semantic Primitives—An Aid to Machine Translation Systems. *Proceedings of Second International Conference on Supercomputing*, 361- 370.

[6] Peterson, J.L. (1980) Computer Programs for Spelling Correction: An Experiment in Program Design. Springer Berlin Heidelberg, 1-129. http://dx.doi.org/10.1007/3-540-10259-0_1

[7] Aoe, J.I. (1989) An Efficient Implementation of Static String Pattern Matching Machines. *IEEE Transactions on Software Engineering*, **15**, 1010-1016. http://dx.doi.org/10.1109/32.31357

[8] Aoe, J., Yamamoto, Y. and Shimada, R. (1984) A Method for Improving String Pattern Matching Machines. *IEEE Transactions on Software Engineering*, **10**, 116-120. http://dx.doi.org/10.1109/TSE.1984.5010205

[9] Stefanakis, G. (2009) Design and Implementation of a Range Trie for Address Lookup. Doctoral Dissertation, TU Delft, Delft University of Technology, Delft. www.repository.tudelft.nl/assets/uuid:a1490c8b-35a3-4f41-a433-0e0d0899c833/thesis.pdf

[10] Shang, H. (1995) Trie Methods for Text and Spatial Data on Secondary Storage. www.cs.mcgill.ca/~tim/cv/theses/shang.ps.gz

[11] Zhao, X. (2000) Trie Methods for Structured Data on Secondary Storage. www.cs.mcgill.ca/~tim/cv/theses/zhao.ps.gz

[12] De La Briandais, R. (1959) File Searching Using Variable Length Keys. *Proceedings of Western Joint Computer Conference*, ACM. 295-298. http://dx.doi.org/10.1145/1457838.1457895

[13] Aoe, J.I., Morimoto, K. and Sato, T. (1992) An Efficient Implementation of Trie Structures. *Software*: *Practice and Experience*, **22**, 695-721. http://dx.doi.org/10.1002/spe.4380220902

[14] Comer, D.E. and Shen, V.Y. (1979) Hash-Binary Search: A Fast Technique for Searching an English Spelling Dictionary.

[15] Pagh, R. (2001) Low Redundancy in Static Dictionaries with Constant Query Time. *SIAM Journal on Computing*, **31**, 353-363. http://dx.doi.org/10.1137/S0097539700369909

[16] Belazzougui, D., Boldi, P., Pagh, R. and Vigna, S. (2009) Monotone Minimal Perfect Hashing: Searching a Sorted Table with O (1) Accesses. *Proceedings of the* 12*th Annual ACM-SIAM Symposium on Discrete Algorithms*, 785-794. http://dx.doi.org/10.1137/1.9781611973068.86

[17] Belazzougui, D., Boldi, P., Pagh, R. and Vigna, S. (2011) Theory and Practice of Monotone Minimal Perfect Hashing. *Journal of Experimental Algorithmics* (*JEA*), **16**, 3-2. http://dx.doi.org/10.1145/1963190.2025378

[18] Botelho, F.C., Pagh, R. and Ziviani, N. (2007) Simple and Space-Efficient Minimal Perfect Hash Functions, Algorithms and Data Structures. Springer Berlin Heidelberg, 139-150.

[19] Amir, A., Farach, M., Galil, Z., Giancarlo, R. and Park, K. (1994) Dynamic Dictionary Matching. *Journal of Computer and System Sciences*, **49**, 208-222. http://dx.doi.org/10.1016/S0022-0000(05)80047-9

[20] Khancome, C. and Boonjing, V. (2014) A New Linear-Time Dynamic Dictionary Matching Algorithm. *Computing and Informatics*, **32**, 897-923.

[21] Chatterjee, S. and Prins, J. (2005) COMP 203: Parallel and Distributed Computing. PRAM Algorithms. Course Notes. www.cs.yale.edu/homes/arvind/cs424/readings/pram.pdf

[22] Xavier, C. and Iyengar, S.S. (1998) Introduction to Parallel Algorithms, Vol. 1. John Wiley & Sons, Hoboken.