A Fast Data Structure for Anagrams

[1]Sk. Mohiddin Shaw, [2]Hari Krishna Gurram,
Department of Mathematics
NarasaraoPet Engineering College,
Narasaraopet, Guntur, Andhra Pradesh, India
Email: mohiddin_shaw26@yahoo.co.in.,
Department of Computer Sciences,
UCEK, JNTU Kakinada, Andhra Pradesh, India
Email: harikrishna553@gmail.com.,

[3]Rama Krishna Gurram, [4]Dharmaiah Gurram,
Department of MCA, NIT Warangal
Email: Ramakrishna.nitw19@gmail.com.,
Department of Mathematics,
Narasarao Pet Engineering College, Narasaraopet.
Email: 10072014nec@gmail.com.,

***Abstract*: -** In this paper, we are presenting a data structure, which stores the given dictionary data in a hash table called PRIME , by using fundamental theorem on Arithmetic to generate a key for each dictionary word, and stores the word in the hash table based on the key. As compared to tree-based techniques PRIME table generates anagram for the given random word in O(1) time, time to construct a PRIME table depends on the number of words in the dictionary. If dictionary has 'n' words then the time to develop the PRIME table is O(n).

Categories and Subject descriptors: Problem solving, search and control methods.

keywords*: Algorithms, performance, Experimentation, fundamental theorem on Arithmetic, Hash map.*

*****

1. INTRODUCTION

An anagram is the result of arranging the letters of a word to produce a new word, using all the letters in given word exactly once (example dgo is arranged as god, dog). Multiple anagramming is a technique used to solve some kinds of crypto programs such as permutation ciphers and transposition ciphers. According to, some historian anagrams originated in 4th century BC. Other resources suggest that Pythorogous, in the 6th century BC used anagrams to discover deep philosophical meanings. Many scientists such as Galileo, Robert Hooke often recorded their results in anagram form to stake their claim on discovery and prevent anyone else claiming the credit. In recent decades, anagrams have become popular in a different role. They are often included in the clues for cryptic crosswords. We are proposing a data structure that works better than tree-based techniques like Anatree to get the anagram for a given word. As compared to Anatree, PRIME table is a simple hash table based data structure that works efficiently.

2. TERMINOLOGY USED

Fundamental theorem of arithmetic states that every integer greater than 1 is either prime itself or the product of prime numbers and the prime factor decomposition of a number is unique. It is also called as unique factorization theorem or unique-prime-factorization theorem.

In computing, a hash map (also hash table) is a data structure used to implement an associative array, a structure that can map keys to values. A hash table uses a hash function to compute an index into an array of buckets or slots, from which the correct value can be found.

3. DATA STRUCTURES

There are number of data structures that support anagrams. The most basic data structure is alphabetic map. Alphabetic map takes the letters of the word and sort them into alphabetical order and produce a mapping from these sorted letters to the word that produced them. Unfortunately algorithms that are developed based on alphabetic map have to sort ever word to find the mapping in the Hash table. Anatree is another data structure for giving anagrams. Anatree is a directed tree which denotes a set of words W encoded as strings in some alphabet A. Internal vertices are labeled with letters from A; edges are labeled with nonnegative integers; leaves contain subsets of W. The efficiency of finding anagram of given word in Anatree is based on the tree construction, since each internal node in the Anatree represents an alphabet and based on the edge weight the tree traverse to the next level. Choosing a vertex label is most challenging problem in construction of Anatree.

4. PRIME TABLE

PRIME table is a Hashmap, each key in the hash map is the product of prime numbers that are related to each character of a given word.

PRIME table is constructed in 2 stages.

Stage 1: Find the key value for a given word.

To find key value for the given word, first all the alphabets in the given language should assigned with weights of unique prime numbers. For every word in the dictionary the key is constructed by multiplying all the weighted values of each character in the given word. Key construction is explained in the below example.

A	B	C	D	E	F	G	H	I	J	K
2	3	5	7	11	13	17	19	23	29	31

L	M	N	O	P	Q	R	S	T	U	V
37	41	43	47	53	59	61	67	71	73	79

W	X	Y	Z
83	89	97	101

Table 1: Index table for English Alphabet:

 ISSN: 2321-8169
2954 – 2956

As shown in table 1 each character in English alphabet is assigned with distinct prime number. English alphabet has 26 characters so we have taken first 26 distinct prime numbers.

For the word: *god*
The key construction is
getKey (*god*) = 17 * 47 * 7 = 5593
Since the prime weight for the letter '*g* ' is 17
prime weight for the letter '*o* ' is 47
prime weight for the letter '*d*' is 7

Step 2: Store the word in hash map based on the key value.

Once the key generated for the given word the word is store in the Hash map based on the key value. There is possibility that more than one word has same key value, in that case key value is pointed to all the words that have same key value.

Example *god* and *dog* has same key value 5593, then HashMap(5593) points to both the strings *god*, *dog*. I.e., HashMap (5593) returns both the words.

4.1 ALGORITHM:

Terminology used in algorithm:

DictFile : File containing all the words

readNextWord() : Reads next word in the given file

getKey() : Returns the key value for the given string

str1.append(str2) : Appends string str2 to the string str1

put(key, string) : Associates the specified string with the specified key in this map.

get(key) : Returns the value to which the specified key is mapped in this identity hash map, or null if the map contains no mapping for this key.

alphaWeight array: Array that has prime number values for each character in the alphabet of given language

findIndexValue() : Returns the prime number weight of given character.

Length() : Returns the number of characters in given string

search() : Returns the anagrams for the given random word

charAt() : Returns a character at particular position the string

HashMap<Integer Key,String Value> dictMap :

A Hash structure with fields key of type integer and value of type string.

```
/* Generate HashTable  for given set of words*/
processDictionary(DictFile)
{
    do
    {
        String = readNextWord(DictFile);
        Key    = getKey(String);
        if((value = dictMap.get(key)) != null)
       {

            value.append(","+string);
            dictMap.put(key, value);
                    }
         else
        {
            dictMap.put(key, string);
                    }

       }while(End Of File reached)
  }

/* Find the value of the alphabet from table/Array Index */
Intger findIndexValue()
{
      return (value of the alphabet from the
      alphaWeight array);
      //value for A is 2 and H is 19
}

/* Return the key value of given string */
Integer getkey(String)
{
            mulValue = 1
            for I = 1 to length(string)
            {
               indexValue =
               findIndex Value (string.charAt(i));
              mulValue = mulValue * indexValue
            }
                    return mulValue
}

String search (findAnagram)
{
            keyValue = getkey (findAnagram)
            return dictMap.get(key)
}
```

4.2 WORKING WITH PRIME TABLE

Let us illustrate the use of PRIME table with an example algorithm. A typical query is to find the anagrams of given string. This can be solved directly with PRIME table. For example to find the anagram of '*ptrulias'* we first find the key value of the string '*ptrulias'*, and retrieve the string mapped with this key value from the Hash Map. Here we are using Hashmap, so to find the Anagram of any given

string will takes constant time, that is, O(1). PRIME table also supports the complex queries.

4.3 SPACE REQUIREMENTS

For PRIME table, let us assume 'N' be the number of words in dictionary. Let 'K' be the total number of keys required to store the 'N' words in hash map. So the total space required for PRIME table is O(N+K).

PRIME table supports solving following queries in constant time.

Anagram: Determine whether a given string of letters has an anagram. For example, the input *dgo* should return true, since it is an anagram of *god*, while *ulias* should return false, since it has no anagram.

All-words: List all words, which use only (but not necessarily all) of the letters of the input string. For example, from the input *FOOT*, we can make *foot, oft, oof, oot, too, of, oo* and *to*. To solve those kind of queries time taken by the PRIME table is equal to total number of combinations of given string. Let 'C' be the total number of combinations of given string, then the time required to get all the anagrams of given word is O(C).

The drawback of PRIME table is it won't support the wildcards efficiently.

Wildcard: Determine whether there is any letter than can be added to the input to produce an anagram. For example, *FX* should return a positive result, because the letter *O* can be added to produce *fox*.

5. Conclusion and Future work

Our primary conclusion is that the PRIME table is a powerful data structure for answering letter level equality and inequality queries. It gives the anagram of given random word in constant time. Its primary disadvantage is it won't give the wildcard of anagram efficiently. We believe that further optimization in this area is fruitful.

REFERENCES

[1] AHO, A. AND ULLMAN, J. "*Foundations of Computer Science"*. W. H. Freeman, New York, 1992, PP 542–545.

[2] AKERS, S. B. "Binary decision diagrams", *IEEE Trans. Comput. 27*, 6, (1978) PP 509–516.

[3] APPEL, A. W. AND JACOBSON, G. J, "The world's fastest Scrabble program", *Comm. ACM 31*, (1988) 572–579.

[4] BOLLIG, B., LBBING, M., ANDWEGENER, I, "Simulated annealing to improve variable orderings for OBDDs". In *Proceedings of the International Workshop on Logic Synthesis*. 5(1995).

[5] CURRAN, K.,WOODS, D., AND RIORDAN, B. O, "Investigating text input methods for mobile phones", *Telemat. Inf. 23*, 1, (2006) PP1–21.

[6] DICKSON, L. E. "*Diophantine Analysis"*. History of the Theory of Numbers Series, vol. 2. Chelsea, New York.

[7] GORDON, S. A, "A faster Scrabble move generation algorithm", *Softw. Pract. Exp 24*, (1994) PP219–232.

[8] SHANNON, C. E, "Prediction and entropy of printed English", *Bell Syst. Techn. J. 30*, (1951) 50–64.

[9] WARD, D. J. AND MACKAY, D. J. C, "Artificial intelligence: Fast hands-free writing by gaze direction", *Nature 418*, (2002) 838–840.

[10] CHARLES REAMS, " Anatree: A Fast Data Structure for Anagrams", ACM Journal of Experimental Algorithmics, Vol. 17, No. 1, (2012).