IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661,p-ISSN: 2278-8727, Volume 18, Issue 4, Ver. III (Jul.-Aug. 2016), PP 41-45
www.iosrjournals.org

Efficient Construction of Dictionary using Directed Acyclic Word Graph

Srinivas.H[1], Amruth.V[2]
[1]*Department of Information Science and Engineering, Maharaja Institute of Technology Mysore, India*
[2]*Department of Information Science and Engineering, Maharaja Institute of Technology Mysore, India*

Abstract: *Implementation of dictionary is a topic on which research is going on since a long time in the search of a better and efficient algorithm both in terms of space and time complexity. Its necessity has increased due to the advent of recent technical advances in the fields of Pattern Recognition, Image Processing, Natural Language Processing and Machine Learning. The matter is of utter importance when it comes to handheld devices where limited memory availability is a crucial factor. From a programmer's point of view, the earlier and the conventional methods used to implement a dictionary were hashing and suffix trees which subsequently failed in order to provide the desired space efficiency. By far the data structure which has proved to be the best for storing a dictionary has been Directed Acyclic Word Graph (DAWG), which merges the common suffixes in addition to the common prefixes of the words. The current method used for implementing it, requires construction of an intermediate Trie which puts a lot of memory constraints and is very time inefficient. In this paper, we describe a new method for creating a compressed dictionary with DAWG using Associative Arrays and concepts of Finite Automata on-the-fly as and when a new word is encountered in the lexicographic order. The method presented in the paper doesn't require construction of an intermediate Trie and hence has a better time complexity. The concept of right language has been crucial to derive the algorithm and is mentioned in the paper. The algorithm is well explained with an example. In the end, the time complexity of the algorithm is analyzed and a small comparison is made in the terms of the maximum memory requirement, for different number of words, for both of the above mentioned methods of constructing DAWG at runtime.*

Keywords: *Deterministic Finite Automata (DFA), Associative Arrays, DAWG, Trie, Right Language, Natural Language Processing*

I. Introduction

Dictionaries are used in various applications mainly in fields of Natural Language Processing (NLP). NLP deals with a huge amount of data and efficient utilization of memory has always been the matter of concern in this field. New algorithms are devised for efficient usage of memory keeping in mind the time complexity.

Dictionary is a collection of words whose efficient storage and retrieval time complexity has always in light for research. There are various algorithms for creating dictionaries [1] using various data structures. One such data structure is Directed Acyclic Word Graph (DAWG) [2].

The Myhill–Nerode theorem [3] states that minimal deterministic automaton of a given language is a unique automaton that accepts the language (excluding isomorphisms) in the minimum number of states among various deterministic automaton. DAWG works on the same principles.

The method explained in this paper uses Associative Arrays to represent DAWG which enhances the searching time complexity from the dictionary.

II. Mathematical Preliminaries

Deterministic finite automaton [3] is a five tuple, used for depicting the transitions from one state to another using some symbols. It can be denoted as 'A' and in the five-tuple notation can be represented as

$A = (Q, \sum, \delta, q0, F)$

where Q: a finite set of states; $\sum$: a finite set of input symbols; δ: a transition function that takes as arguments as state and an input symbol and returns a state. If q is a state, and 'a' is an input symbol, then $\delta(q,a) = p$ is a transition leading to any state $p \in Q$, q0: a start state; F: a finite set of accepting states. F is a subset of Q

Associative Array is an array which associates a value to a key and is represented as a tuple (key, value). The key is unique and appear only once. The values associated with a key define the key and is accessible with the given key.

A DFA can be represented by an Associative array. Let L = L(M) for some DFA, $A = (Q, \sum, \delta, q0, F)$. The transition function $\delta(q, a) = p$ is mapped to the associative array as

(q, concatenation(a, p))

and then the similar transition is taken from state p.
key(0) is defined as q0 and key(1) is defined as F (since only one final state is required in this case). Thus a string, w is in L if and only if *w* starts at key(0) and ends at key(1), i.e., there exists a series of transitions that takes such that (key(0), (*w*, key(1))
The language that is followed from a given tuple (key) is the right language of that key and defines the minimal finite automata.

III. Construction Of Dictionary

Before discussing the algorithm, the implementation of dictionary using tries is highlighted. A trie is a tree data structure in which transitions can be viewed as root being the start state and rest of all the states as final states. Trie constructs separate node for each of the alphabet of the words in the dictionary. Transitions for each of the word to declare whether it is there or not has separate and individual sub tree. But, the major disadvantage of constructing a trie is it occupies a lot of space and each time we insert a new word, a different transition is to be created. So to avoid such problems we usually try to compress the trie which can done using Hopcroft's algorithm of DFA minimization, but the problem with this method is that it takes a lot of space to construct the intermediate trie which may not be available at the first place in a few situations.

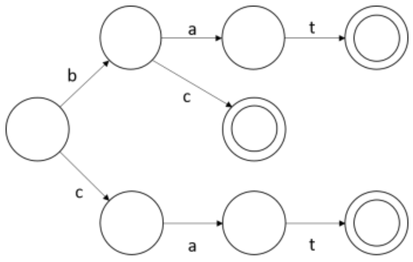

Fig.1. Trie for bat, bc and cat

Creation of Associative Arrays happens dynamically and optimizations are done and the state of dictionary at a particular point represents the minimal finite automata for the set of words occurred till now (as shown in Fig 1).

The sorted wordlist is the prerequisite for the construction of dictionary from the given method. The associative array created correctly searches for a word in the dictionary. The searching starts from key(0) and on the basis of the transition letter next tuple is found from the value of the given key. The series of transitions are carried out and if the end tuple is key(1) then the word exists in the dictionary else it doesn't. The key(1) doesn't contain any value as it is correspondent to the final state (Table I).

Table I: Associative Array representing dictionary of words bat, bc and cat.

KEY	VALUE
0	b2, c4
1	
2	a3, c1
3	t1
4	a3

The main factor of optimization occurs at the suffixes. Two keys are same if their values are exactly the same and hence the keys can be considered as one. This practice can be done on-the-fly and hence a minimized dictionary is obtained at the insertion of every string solving the problem of memory constraints caused due to construction of a trie.

IV. Algorithm

The algorithm demonstrates the construction of the Associative Array that would represent the dictionary (as shown in Fig 2).

```
Dictionary = NULL
New_Entries = NULL
tuple = 2
do there is another word:
if word is prefix of next word:
word = concatenate(word, 'µ') start at key(0)
while next character of word:
if key(character is present):
```

```
last_key = next key concatenated with character
go to last_key else:
jump out of loop
insert_into_table(rest of the word, last_key)

insert_into_table(word, last_key): do there is another
character:
if last character:
insert into New_Entries concatenate(character, 1)
jump out of loop
else:
insert into New_Entries concatenate(character, tuple)
jump to newly generated key
tuple = tuple + 1 optimize(New_Entries)
append New_Entries to Dictionary New_Entries =
NULL

optimize(New_Entries):
do there is another key k in New_Entries:
if k.value == Dictionary(any key k1).value change
all values of all keys having k to k1
delete k from New_Entries tuple = tuple – 1
```

Fig 2: Proposed Algorithm

For every next word, its prefix that is already present in the Dictionary is found out and the rest of the word is sent to be inserted into the table. The variable tuple represents the key numbers. insert_into_table() starts appending characters into New_Entries which is another Associative Array that contains newly entered values. If it's the last character then 1 is concatenated since key(1) is considered as the final state. After entering the (key, value) pairs the New_Entries is sent for optimization. If a key k of New_Entries is found to have to the same value as that of a key k1 in then the values of all the keys in New_Entries is changed to k1, thus the minimization is achieved. The minimized New_Entries is then appended to Dictionary to insert the new word. Thus, minimization is achieved at the insertion of every word that is inserted into the dictionary.
µ is added to the word if it's a prefix of the next word so as to avoid ambiguity in the DFA. For example, if bat and bats constitute the dictionary then its Associative Array would be as in Table II.

Table II: Associative Array of bat and bats.

Key	Value
0	b2
1	
2	a3
3	t1, t4
4	s1

It can be seen that there is ambiguity at key(3) which takes the same character to different keys (tuples). Hence a special character µ is used to overcome this ambiguity. µ is a character which is not present in the character set of the language, i.e., µ doesn't belong to ∑.

Table III: Associative Array of bat and bats after insertion of µ to resolve ambiguity.

Key	Value
0	b2
1	
2	a3
3	t4
4	s1, µ1

After the concatenation µ the new Associative Array would be as in Table III and thus the ambiguity is resolved.

V. Analysis Of The Algorithm

The proposed algorithm runs at an time complexity of O(nm) where n is the number of words and m is the number of characters in each word. Since number of characters in a word is negligible as compared to number of words in a large dictionary, the time complexity essentially becomes O(n) which is better than trie minimization whose time complexity is O(nlogn) using Hopcroft's DFA minimization algorithm.

The searching time complexity is O(mp) or O(mp+1) depending on the fact if µ is added or not. Here p is the number of member values of a key. Thus, the complexity is essentially a constant factor if a large dictionary is considered. The proof of the above statement is the fact that it takes O(1) to access a tuple of an array and O(p) to traverse across the member values of a key.

VI. Results

The proposed algorithm even creates lesser number of nodes as compared to trie thus overcoming the memory constraints caused by the construction of the trie. A node for an Associative Array is its tuple (key).

Table IV: Comparison of results obtained.

Number of Words	Number of nodes in Trie	Number of tuples in Associative Arrays
100	792	512
200	1362	747
500	3225	1456
1000	6249	2493
2000	11457	3976
4000	21300	6354

From the results shown in Table IV, it can be inferred that the memory required for dictionary using Associative Arrays if far lesser than that required by a Trie and hence its more optimal where space is the main criteria.

VII. Applications

Representing large vocabularies for use in NLP applications: It is a space as well as time efficient method of storing large vocabularies. This maximizes portability of the dictionaries to devices which are space as well as time constrained (due to low processor speeds) to handheld devices and real time systems.

Online ECG lossless compression for the cases where we have limited bandwidth: It reduces data size by eliminating redundant data and lossless compression technique allows for the recovery of data in its intact form. This is particularly useful in the areas of limited bandwidth. If the data further needs to be encrypted for security purposes, this method ensures efficient utilization of available bandwidth.

Finding longest common factor of two words: As the common nodes between the words are shared to the maximum extent possible, we can find the longest common factor of two words. Further, because of the linear complexity, this enables fast searching mechanism for the words when compared to previous algorithms. We can extract homophones in a language using this property of DAWG.

Spell checking and Predictive text: It is very convenient method for spelling checking and prediction of text while typing by showing the various possible options to the end user. This also minimizes the possibility of errors creeping in due to phonetic resemblance in the words.

Thesaurus: When given a word, a simple thesaurus may appear to be like for word "see"
Definition - verb (used with object), saw, seen, seeing: to perceive with the eyes; look at, to scan or view, especially by electronic means
Verb - detect, examine, identify, look, look at, notice, observe, recognize, regard, spot, view, glare, glimpse, eye, catch sight of, peep, and peek.

DAWG may be used for generating thesaurus in a compact manner with some additional data structure like lists to represent the words as vectors.

VIII. Conclusion And Future Work

The paper presents a novel algorithm of creating a DAWG which serves as a useful tool in the areas where compact representation of large vocabularies with direct access is required. The algorithm needn't create a Trie as required in the previous methods of implementation and is indeed has a reduced time complexity in terms of constructing it but requires a sorted list of words.

In the future, we would like to work on the deletion of words from DAWG at runtime without restructuring the existing dictionary again from scratch. This would prove helpful in the cases where dynamic deletion of words is required and would save memory as it would allow the changes to be incorporated without creating another dictionary for deleted words. Other area of improvement would be create an algorithm which can create DAWG from an unsorted list of words.

Another interesting area is to work the algorithm for local Indian languages like Kannada, Marathi, Hindi and store their pronunciations as well in the structure along with the words. This would allow text to speech conversion of the words and hence a large text at increased efficiencies.

References

[1]. Watson, Bruce W. 1993a. A taxonomy of finite automata construction algorithms. Computer Science Note 93/43, Eindhoven University of Technology, The Netherlands. Available at www.OpenFIRE.org.

[2]. Thulasiraman, K. Swamy, M. N. S. (1992), "5.7 Acyclic Directed Graphs", Graphs: Theory and Algorithms, John Wiley and Son, p. 118, ISBN 978-0-471-51356-8.

[3]. Hopcroft, John E. and Jeffrey D. Ullman (1979) “Introduction to Automata Theory, Languages, and Computation” Addison-Wesley, Reading, MA.

[4]. Jan Daciuk, StoyanMihov, Bruce W. Watson, Richard E. Watson (2000) “Incremental Construction of Finite Automata”Volume 26, Number 1, Association for Computational Linguistics.

[5]. Cláudio L. Lucchesi, Tomasz Kowaltowski (1992) “Applications of Finite Automata Representing Large Vocabularies” Relatório Técnico DCC – 01/92.