UACEE International Journal of Advances in Computer Science and its Applications – IJCSIA
Volume 3 : Issue 2 [ISSN 2250 – 3765]
Publication Date : 05 June 2013

A Novel and Efficient Data Structure to Facilitate Dictionary Search using Wildcards

Aloke Kumer Saha
Department of Computer Science and Engineering
University of Asia Pacific
Dhaka, Bangladesh
aloke71@yahoo.com

Khandker Tafiqul Islam
Department of Computer Science and Engineering
University of Asia Pacific
Dhaka, Bangladesh
kh.tafiqul.islam@gmail.com

Sheikh Muhammad Sarwar
Department of Computer Science and Engineering
University of Liberal Arts
Dhaka, Bangladesh
sheikh.sarwar@ulab.edu.bd

Bindu Rani Das
Department of Computer Science and Engineering
University of Asia Pacific
Dhaka, Bangladesh
supriya.rani32@yahoo.com

Md Ashrafujjaman Mondal
Department of Computer Science and Engineering
University of Asia Pacific
Dhaka, Bangladesh
babu_ziz007@yahoo.com

Sofura Akhter
Department of Computer Science and Engineering
University of Asia Pacific
Dhaka, Bangladesh
laj.cseuap@gmail.com

***Abstract*— In this paper, a novel and efficient data structure named as 'Augmented Trie' has been proposed that can store a large collection of English words and search them efficiently. The data structure has been specially designed in a way to facilitate the search for strings with wildcard characters. Even though the memory requirement for constructing 'Augmented Trie' is higher than simple trie, it does not become significant as memory allocation has been performed intelligently using bit masking. By experimental results we show that the proposed method of finding patterns with wildcard characters improves over the existing one by 13.5% (maximum).**

***Keywords*—data structure, algorithm, wildcard search.**

I. Introduction

Wildcard search has become an inevitable feature in search engines and databases to facilitate the users, who search with patterns [1] [2] [3]. According to the problem definition, given a list of strings and a wildcard pattern a user would like to know how many strings are there those match the wildcard pattern [4]. A wildcard is a character that can be used as a substitute for one or many number of unknown characters in string search and as a result it increases the flexibility and efficiency of searches. Wildcard searching has been applied in the MSN search index [6] and ranking algorithm for listing the most popular results. Google is also providing wildcard searching [5]. Moreover, it has also been very useful in structured peer-to-peer networks and distributed environments [5]. As it needs less time and typing to find any word, it is becoming more and more popular day by day. So, efficient data structures and reliable algorithms are needed for wildcard searching. This topic has been quite an important subject of research in recent years. Most importantly, research is going on about discovering fast and suitable algorithms, which tend to use data structures those take less space in memory.

There is another interesting research area of recognizing words from human gesture, where wildcard search can be really useful. When a user wants to input a word using gestures, the image of the whole word can be captured after the sequence of gestures is provided. But some characters of the word can be really hard to recognize from that image and they remain unknown. Now, these characters can be treated as wildcards and by combining with the recognized characters from the image a pattern can be formed. This pattern can be used as wildcard search string to find all the possible words in the dictionary, those could be the intended inputs of the user. Then the user can choose any of the possible words. Even a ranking of the words can be provided for the user.

In this paper, we propose a method for textual dictionary search, where the search string contains wildcard characters. Our method is fast and intelligent enough to reduce the machine time and find all the words in the dictionary given a pattern. We augmented the classic trie data structure by adding additional information for each node and that helped to search the

dictionary efficiently. We also encoded the additional information using bit masking, which consumed little amount of memory for this purpose. By performing our experiment, we found that it takes less time compared to the linear scanning method and the existing trie based method [6].

The paper is divided into 6 sections. The second section of the paper explores the background ideas behind wildcard searching. The third part describes our proposed solution in detailed manner along with our 'Augmented Trie' data structure. In the fourth section we include statistics regarding some test run after the implementation of our method. Finally, in the fifth section we describe the fields where it may be applicable, and how far the idea can be extended. Our new algorithm is robust and fast in practice.

II. Background

A. *Wildcard Search*

Searching a string for a specified pattern is called wildcard search [7]. A wildcard may be an asterisk(*) or a question(?) mark. An asterisk(*) means empty or multiple unknown characters and a question(?) mark means a single unknown character. They can be substituted using possible letters to make up a word. The question mark (?) character matches the word that can be formed by replacing a single character. For example "hum?n" or "huma?" will produce the result "human". An asterisk(*) means multiple or zero replaceable characters. For example "humanis*", this will produce the results like "humanish" , " humanism" , "humanist" , "humanistic" , "humanistical" , "humanistically". The wildcard term can be used in the as a prefix "*umanism" or as a suffix like "humanis*". Multiple numbers of wildcards and a mixture of both types of wildcards are also allowed as per our problem definition. An example of this can be "hu*an*?m".

This type of problem can be solved by constructing a DFA (Deterministic Finite Automaton), linearly matching all the strings from a list against the DFA and finding the possible matches [8]. Many programming languages and libraries have built in support for these kinds of pattern matching. They give correct result, but lacks efficiency when time complexity is considered.

B. *Trie*

Trie is a tree based data structure that is used to store a list of strings by using less memory. It was invented by Edward Friedkin, who defined it as an associative array for elements which are usually strings [9].

Trie reduces the cost of memory and number of searches to find a specific word or a string with wild cards from a database of millions of words. When implemented in straightforward manner, it is best for searching for patterns like "Hel*" or "He???". However, when a wildcard term is used as prefix such as "*ing", it does not work efficiently because a lot of word ends with "ing". In this case, the searching resorts to checking all root-to-child paths until the string "ing" is found. Fig. 1 shows the construction of a trie for a small set of words.

C. *Bit masking*

Bit masking is a technique that by which we can work on individual bit of any integer [10]. We can set or clear a bit. We also can toggle a bit. These are bone by bit wise operators (AND, OR, XOR). Bit masking reduces the use of memory.

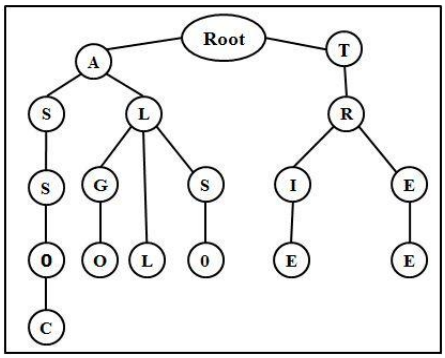

Figure 1: The construction of atrie using words "Assoc", "Algo","All", "Also", "Tree" and "Trie".

III. Proposed Method

A. *Searching Procedure*

In our system, the data structure for storing words and algorithm to search for a specific pattern is devised in such a way that:

1) The space needed to store information is minimal
2) Search space in pruned effectively and branching factor is reduced when searching for a pattern with wildcards.

To achieve these effects we have added the following extra information for each node in the trie to create an augmented trie:

1) A Boolean mark is stored in a node to denote if a word can be constructed following the path from root to this node.
2) An array of *n* integers, where each of the integers is 32-bit. Now, each bit of an integer denotes the presence or absence of a character. We can set each bit by using the bit masking

operation mentioned in background section. As there are 26 characters in English language, a 32-bit integer can be considered enough to store the state of each letter. Now, if the j^{th} bit of i^{th} integer is set as 1 in the in the integer array, then it can be stated that the j^{th} character is present at depth i below the current node.

Our searching technique can be described as below:

1. When the dictionary is loaded, the words are loaded in the trie. At the time of backtracking, the integer array in each node is updated with the information for each character c. If c is located in any of its children's sub tree in depth i, the bit corresponding to letter c if integer i is set. From Fig. 2 it can be observed that for the expanded node 'L' three integers from the integer array are shown. By inspecting the content of a[1], it can be observed that the 7^{th}, 12^{th} and 19^{th} bits are set as 1, while the others are set as 0. Alphabetically the 7^{th}, 12^{th} and 19^{th} characters are 'G', 'L' and 'S' respectively. We can easily conclude that at one depth below 'L', there are nodes containing 'G', 'L' and 'S'.

2. The augmented trie can be searched for patterns with fixed length wildcards, such as '?'. Suppose a pattern like '??SOC' has to be searched from the augmented trie shown in Fig. 2. Now, if the integer array in first level node containing character 'A' is examined, it will be found that there is a node containing character 'S' at two level depth from character 'A'. But there is no node containing character 'S' at two level depth from the first level node containing character 'T'. So, this branch can be pruned easily and here we can see the efficiency of our augmented trie.

3) When multiple character wild card (*) is used, we start scanning from first character till a wildcard character is found. Let's assume that the i^{th} character is the wild card character. Suppose, we have a pattern like 'AL*O' and we are searching the trie shown in Fig. 2. So, we already have traversed the nodes 'A' and 'L' and inspecting the childs of 'L'. Now, there are only two children 'G' and 'S' who have 'O' as their descendant. We can easily be sure of that by checking the set bits of 'G' and 'S'. But, after inspecting the 'L' nodes integer array, we can conclude there is no 'O' node as its descendant. Hence, we can prune this branch. Now, if there is no known character after the wild card character '*', we will have to scan through all the sub nodes from current node until we reach to all ending sub nodes. This will happen if we have to search for a pattern like 'AL*' from the trie shown in Fig. 2. In this case we will have to traverse all the nodes below 'L' and the resultant patterns will be 'ALGO', 'ALL' and 'ALSO'.

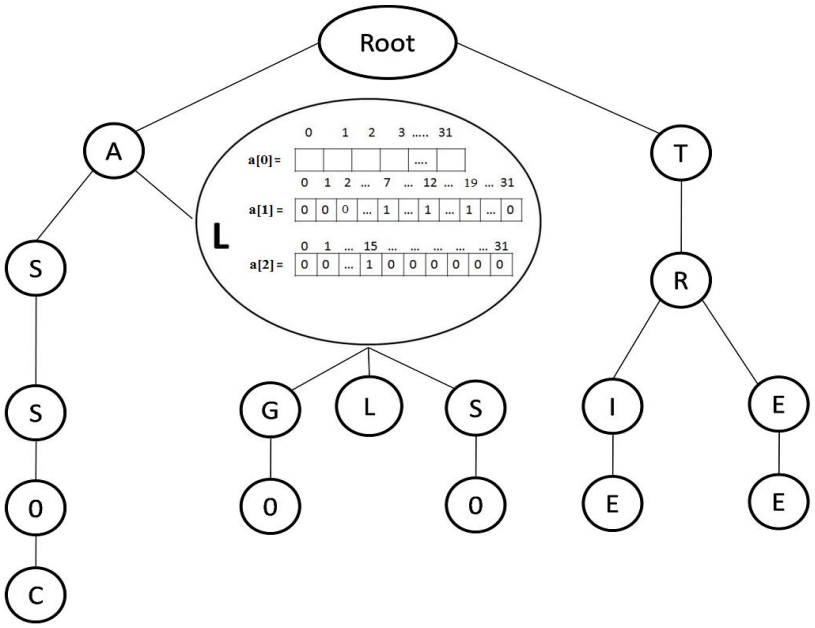

Figure 2. Representation of our proposed 'Augmented Trie'

B. *Detailed data structure*

We can find a given word efficiently from normal "trie". But for wildcard search algorithm, normal "trie" cannot be the most efficient choice. That's why we have to store some more information in each node of the "trie" and create our "Augmented Trie". Along with the character of current node and the pointer to all the child nodes, we have stored a Boolean variable to mark a node as ending of any string. We have also used an integer array of length 20 to store the information of the characters those are present in all the descending nodes of the current node. The i^{th} index stores the information of the characters those are at i^{th} depth from a specific node in the trie. That helps us to find and any types of wildcard patterns in an efficient manner. The structure of a node is as follows:

```
struct node
{
   char c;
struct node* child[26];
int a[20];
bool en;
   node(char c)
   {
      this->c=c;
      for(int i=0;i<26;i++) child[i]=0;
      for(int i=0;i<20;i++) a[i]=0;
      en=false;
   }
   node()
   {
      c=0;
      for(int i=0;i<26;i++) child[i]=0;
      for(int i=0;i<20;i++) a[i]=0;
      en=false;
   }
};
```

IV. Experimental Result

We implemented and tested our method on a pc containing Intel® core™ 2 Quad CPUQ8400 @2.66GHz with 2 GB RAM. We used a text dictionary [11] of English words those are commonly used in everyday language including some common scientific words. Table 1 shows the timing comparison of our proposed algorithm and existing best algorithm [6]. In order to test our method we randomly generated 5000, 10000 and 20000 search queries and calculated the time for our proposed algorithm and the existing best algorithm to search the trie built using [11]. It can be easily observed that the minimum improvement is 11.5% and the maximum improvement is 13.5% considering time. From Table II it can be observed that our data structure and algorithm performs well for individual types of search queries. In this experiment we used dictionaries of different word size. We also showed the amount of memory (in MB) that was consumed to store the dictionary. It can be said that even if we added some extra information for each node in the trie, the memory usage is not that much high. So, we can say that wildcard searching with our proposed strategy brings a notable improvement. This strategy does not affect the full word search. For full word search our algorithm has the time complexity of a traditional trie.

TABLE I. COMPARISON OF PROPOSED AND EXISTING ALGORITHMS IN TERMS OF TIME (IN SECONDS)

Number of Queries	Time		
	Proposed	Existing [6]	improvement
5000	1.73	2	13.5%
10000	4.36	5	12.8%
20000	7.96	9	11.5%

V. Conclusions

The proposed 'Augmented Trie' is efficient in wildcard searching, as it makes the use of simple yet elegant logic to reduce the search space. By carefully limiting the maximum distance between two characters during searching using the maximum distance between those characters in any words in reference dictionary, we can prune our search space quite further. Its simplicity can be exploited in smart phones, where memory can be extended and memory addressing might be slower. As the data structure is not quite complicated to maintain; and once built, it stays constant overtime, leading to the opportunity of using it in multithreaded applications.

VI. References

[1] D. J. Byrne, J. M. McConaughy, S.-B. Shi, C.-L. Shu, and T. M. Tran, "Reverse string indexing in a relational database for wildcard searching," Mar. 6 2001, uS Patent 6,199,062.

[2] R. M. Lane, "Method and apparatus for facilitating wildcard searches within a relational database," Apr. 29 2003, uS Patent 6,556,990.

[3] S. A. Friedberg, "Lock-free wild card search data structure and method," Dec. 9 2003, uS Patent 6,662,184.

[4] P. Clifford and R. Clifford, "Simple deterministic wildcard matching," *Information Processing Letters*, vol. 101, no. 2, pp. 53–54, 2007.

[5] Y.-J. Joung and L.-W. Yang, "Wildcard search in structured peer-to-peer networks," *IEEE Trans. Knowl. Data Eng.*, vol. 19, no. 11, pp. 1524–1540, 2007.

[6] X. Zhou, Y. Xu, G. Chen, and Z. Pan, "A new wildcard search method for digital dictionary based on mobile platform," in *Proceedings of the 16th International Conference on Artificial Reality and Telexistence–Workshops*, ser. ICAT '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 699–704. [Online]. Available: http://dx.doi.org/10.1109/ICAT.2006.19

[7] (2006, July) How to use wildcards, by the linux information project (linfo). The Linux Information Project. [Online]. Available: http://www.linfo.org/wildcard.html

[8] M. Becchi and P. Crowley, "Extending finite automata to efficiently match perl-compatible regular expressions," in *Proceedings of the 2008 ACM CoNEXT Conference*, ser. CoNEXT '08. New York, NY, USA: ACM, 2008, pp. 25:1–25:12. [Online]. Available: http://doi.acm.org/10.1145/1544012.1544037

[9] D. E. Willard, "New trie data structures which support very fast search operations," *J. Comput. Syst. Sci.*, vol. 28, no. 3, pp. 379–394, Jul. 1984. [Online]. Available: http://dx.doi.org/10.1016/-0022-0000(84)90020-5

[10] K. Matz. (2006, June) Atrevida game programming tutorial. Atrevida Game Programming Tutorial. [Online]. Available: http://-atrevida.comprenica.com/atrtut03.html

[11] K. Atkinson. (2010, December) Kevin atkinson. Sourceforge. [Online]. Available: http://wordlist.sourceforge.net/

TABLE II. COMPARISON OF PROPOSED AND EXISTING ALGORITHMS FOR SOME SPECIFIC SEARCH STRING PATTERNS IN TERMS OF TIME (IN MILLISECONDS)

Word Collection Size	Memory (MB)	Search time for pattern 'a*b*c' (in milliseconds)		Search time for pattern '*ing' (in milliseconds)	
		Proposed	Existing [6]	Proposed	Existing [6]
50000	23.08	2	10	3	10
100000	46.55	4	10	6	20
200000	93.61	5	20	9	20
236983	111.07	7	40	10	50