International Journal of Progressive Sciences and Technologies (IJPSAT)
Vol. 2 No. 2 April. 2016, pp. 61-64.

http://ijpsat.ijsht-journals.org

An Auto-completion Algorithm Using Conditional Probability

Anuj Vaijapurkar[1], Rishon Patani[1] and Vaibhav Kashyap Jha[1]

[1]Department of Computer science, Vellore Institute of Technology,
Vellore, India

ABSTRACT: - In this paper, we present an algorithm to auto-complete words that work on user-input by suggesting the highly probabilistic words to the user. Our algorithm uses trie Data structure we use this trie for efficient suggestion of words to the user. Our algorithm implements trie efficiently and modify its structure quickly and correctly every time a user enters a word. Typically browsers implement this feature by caching a fixed number of queries, previously entered by the user on the client side. Our algorithm can be used as an offline model for heavy user-input interaction interfaces. Typically browsers implement this feature by caching a fixed number of queries, previously entered by the user on the client side.

KEYWORDS: Graph theory, Conditional probability, Suggestion algorithm, Auto-complete algorithm.

1. INTRODUCTION

Auto-completion or auto-suggestion is a widely use used prominent technique by almost all the major search engines, editors that works on the user-input. After a new character is entered in the editor, it filters suggestions that match the updated word and suggest the top-ranked words.

Text-input interfaces have been undergoing magnanimous changes since the past decade many search engines, editors today suggest auto-completions or suggestions as you type. Following a new character entered in the editor, it filters suggestions that match the updated word and suggest the top-ranked words. The primary step is filtering which is often facilitated by using data structures such as prefix-trees or tries that allow efficient lookups by prefix matching. The whole concept of automatic completion of words or queries has become highly pervasive. An auto-completion mechanism prompts the user with a set of suggestions, each of which represents a suffix or completion of the user-input. Thus auto-completion allows the user to not only avoid unnecessary typing, but also save a lot of time. It has also found its applications in input fields such as file location fields, email address and URL address bars. Along with it, applications such as dictionary-based word or phrase/query completion are also driven by auto-complete algorithms. With the recent technological advancement in the client-server model paradigms like AJAX (Asynchronous JavaScript and XML), users have increased expectations about the applications of auto-completion systems.

It is important to note that auto-completion is only effective if it appears “instantaneous” in the human time-scale, which has an upper bound of 100ms. For single word completion, typical techniques involve building a dictionary of all words as well as possibly coding this as a trie, with each node representing one character and each root-leaf path depicting a word. Such methods should be refrained from directly using in the multiword case because a finite dictionary of multi-word phrases cannot be constructed. Even if we limit the length of the phrases, we would still be in need of an 'alphabet' comprising all possible words. The relative size of such a dictionary would be several orders of magnitude larger than just for a single word case. It is therefore important to remember that auto-completion is useful only when suggestions offered are quick and correct (in that they are selected by the user). Also offering inappropriate suggestions is equally bad to offering no suggestions at all. It distracts the user and makes him anxious.

The upcoming section discuss the method and algorithm we present for auto-completion, its advantages and other industrial applications.

Corresponding Author: Rishon Patani (rishon1313@gmail.com)

2. ALGORITHM

Construct a trie T by inserting keys in the node and store its values. Following a new character entered by the user, filter suggestions that match the updated word and suggest the top results. After the word is added to the trie, a flag is used to represent successful completion of the process. Repeat the above procedure of adding the words to the trie until we encounter the longest word in the dictionary or flags for all words have been set.

Traverse the trie by comparing the corresponding characters of the words in the dictionary in order to return the most probable words the user would enter. Choose the nodes with higher edge costs and keep updating every time a word is entered.

Edge_Cost = (Number of words starting with a particular letter) /(Total number of words in the dictionary).

If the node is reached where the complete word is found, suggest the word to the user and set the flag, which denotes the word has been added to the dictionary. New words other than those in the language used by the user will also add in the trie and updated every time the user inputs the word. It will self-learn and suggest the user with the desired words during word/query completion. Updating would take place during two major instances Word typed by the user matches the word in trie. Word typed by the user does not match with the word in trie.

3. PSEUDO CODE

We will be given with the set of words represented as WORD_LIST. The word in the list will be inserted into the trie using INITIALIZE_TRIE (WORD_LIST). Whenever user will enter word the trie will be traversed using TRAVERSE_TRIE(TRIE,WORD,LEVEL) and most probable word will be suggested to the user. When the user enter the new word or uses the word available in WORD_LIST INSERT_TRIE(TRIE,WORD,LEVEL) function will get called to improve the accuracy of the algorithm. The sample procedures of the above listed functions are as shown below.

```
INITIALIZE_TRIE(WORD_LIST)
BEGIN
    TRIE={}
        FOR WORD in WORD_LIST
            INSERT_TRIE(TRIE,WORD,0)
END

TRAVERSE_TRIE(TRIE,WORD,LEVEL)
BEGIN
    IF TRIE[WORD[LEVEL]] != EW and LEVEL<=WORD.  LENGTH
        THEN
    TRAVERSE_TRIE(TRIE[WORD[LEVEL]],WORD,LEVEL+1)
    END
    ELSE IF TRIE[WORD[LEVEL]]=EW and LEVEL=WORD.  LENGTH
        THEN
            PRINT "Nothing to suggest"
        END
    ELSE
        THEN
            Follow path recursively where edge cost is maximum
        END
```

```
END
  INSERT_TRIE(TRIE,WORD,LEVEL)
  BEGIN
      IF TRIE[WORD[LEVEL]] exist and LEVEL<=WORD.  LENGTH and EOQ=false
            THEN
                  BEGIN
                        Change the edge cost at LEVEL
            IF LEVEL+1 = WORD.  LENGTH
                  THEN
                  EOW :=true
            ELSE
                  THEN
                  EOW:=false
                  INSERT_TRIE(TRIE[WORD[LEVEL]],WORD,LEVEL+1)
                  END
      ELSE
            THEN
                  BEGIN
                        Add the vertex into trie at LEVEL
                        Change the edge cost at LEVEL
            IF LEVEL+1 = WORD.  LENGTH
                  THEN
                  EOW :=true
                  TRIE[WORD[LEVEL]]=EW
            ELSE
                  THEN
                  EOW:=false
                  INSERT_TRIE(TRIE[WORD[LEVEL]],WORD,LEVEL+1)
                  END
  END
```

Here EOW is a flag representing End of the word.

4. FUTURE WORK

We look forward to extend this work to take advantage of efficient term and query matching methods to tailor suggestion lists for individual users using appropriate probabilistic models.

5. CONCLUSION

The probabilistic trie model for auto-completion ensures efficient traversing and reducing the cost required to traverse the nodes using conditional probability. This model is well suited for offline editors and can also be used in interactive applications which require fast suggestions.

ACKNOWLEDGEMENT

The authors wish to thank Prof Yamuna M. for guiding us throughout the paper.

REFERENCES

[1] K. Kukich. Technique for automatically correcting words in text. ACM Computing Surveys (CSUR), 24(4):377–439, 1992.
[2] C. MacArthur. Word Processing with Speech Synthesis and Word Prediction: Effects on the Dialogue Journal Writing of Students with Learning Disabilities. Learning Disability Quarterly, 21(2):151–166, 1998.
[3] G. Manku and R. Motwani. Approximate frequency counts over data streams. Proceedings of the Twenty-Eighth International Conference on Very Large Data Bases, 2002.
[4] L. Paulson. Building rich web applications with Ajax. Computer, 38(10):14–17, 2005.
[5] "Autocomplete 0. 0. 104 : Python Package Index". *Pypi. python. org*. N. p. , 2016. Web. 18 Feb. 2016.
[6] E. M. McCreight. A space-economical suffix tree construction algorithm. J. ACM, 23(2):262–272, 1976
[7] Y. Liu. Learning to rank for information retrieval. Foundation and Trends in Information Retrieval, 3(3):225–331, Mar. 2009. ISSN 1554-0669.
[8] Shokouhi and K. Radinsky. Time-sensitive query autocompletion. In Proc. SIGIR, pages 601–610, Portland, Oregon, USA, 2012. ISBN 978-1-4503-1472-5.
[9]J. Teevan, S. T. Dumais, and E. Horvitz. Personalizing search via automated analysis of interests and activities. In Proc. SIGIR, SIGIR '05, Salvador, Brazil, 2005. ISBN 1-59593-034-5.
[10] Weber and C. Castillo. The demographics of web search. In Proc. SIGIR, pages 523–530, Geneva, Switzerland, 2010. ISBN 978-1-4503-0153-4.
[11]UMLSKS SUGGEST: An Auto-complete Feature for the UMLSKS interface using AJAX,Anantha Bangalore, Allen Browne, and Guy Divita.
[12] L. Zhang, T. Tran, and A. Rettinger. Probabilistic Query Rewriting for Efficient and Effective Keyword Search on Graph Data.
[13] S. Tata, R. Hankins, and J. Patel. Practical Suffix Tree Construction. Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August, pages 36–47.
[14] B. Davison and H. Hirsh. Predicting sequences of user actions. Notes of the AAAI/ICML 1998 Workshop on Predicting the Future: AI Approaches to Time-Series Analysis, 1998
[15] A. Nandi and H. V. Jagadish. Effective phrase prediction. In Proc. VLDB, pages 219–230, Vienna, Austria, 2007.
[16] Reda, Y. Park, M. Tiwari, C. Posse, and S. Shah. Metaphor: a system for related search recommendations. In Proc. CIKM, pages 664–673, Maui, HI, 2012. ISBN 978-1-4503-1156-4.
[17] 102–113, 2001. [9] M. Farach. Optimal suffix tree construction with large alphabets. Foundations of Computer Science, 1997. Proceedings. , 38th Annual Symposium on, pages 137–143, 1997.
[18] Gregory Smits, Olivier Pivert, et al. AGGREGO SEARCH: Interactive Keyword Query Construction.