International Journal of Advanced Technology in Engineering and Science
Vol. No.5, Issue No. 04, April 2017
www.ijates.com
ijates
ISSN 2348 - 7550

SPELL CHECKER AND STRING MATCHING USING BK-TREES

Shridhar Shah

Department of Computer Science and Technology Institute of technology

Nirma University, Ahmedabad (India)

ABSTRACT

*A BK-tree or Burkhard-Keller is a tree information structure specific to list information in a metric space. A metric space is basically a game plan of articles which we outfit with a separation work d(a, b) for each combine of segments (a, b). This separation work must fulfill a game plan of aphorisms with a particular ultimate objective to promise it's overall conduct. BK - Tree or Burkhard-Keller Tree is utilized to choose every one of the components of a settled set, that are like the inquiry component. The innocent way to deal with locate every single nearest component will be to contrast the question component and every single other component of the settled set, accepting that correlation take consistent time i.e. O(1) we will get brings about O(N) where N is the quantity of components in settled set, however in the event that we utilize BK-tree, we can significantly decrease this opportunity to O(L1*L2*log N) where L1 is normal length of component in our settled set and L2 be the length of question component[3].*

Keywords: BK-tree, Edit distance, Levenshtein Distance

I. INTRODUCTION

BK-Trees or Burkhard-Keller Trees are a tree-based data structure worked for quickly discovering close matches to a string, for example, as used by a spelling checker, or while doing a "fuzzy" sweep for a term. The fact of the matter is to return, for example, "hall" and "help" ought to be yield when we examine for "hell". without BK-Tree this issue must be unraveled by brute-force technique procedure where we would have contrasted the inquiry component and each other component in the settled set however with the utilization of BK tree we can up significantly diminish the looking circumstances. Before we can characterize BK-Tree to inquiry a component we require to characterize an arrangement of preliminaries. To list and hunt our settled set, we will characterize a few tenets to look at the string[1]. The standard methodology for this is the Levenshtein Distance, which takes two strings, and returns a number which will give the base number of transformations(i.e. addition, cancellation, and substitutions) to be connected on one component keeping in mind the end goal to change into another component.

Presently we specify some valuable certainties about the Levenshtein Distance: It shapes a Metric Space. Put essentially, a metric space is any relationship that sticks to three central criteria:

1. $d(x, y) = 0 <-> x = y$ (If the separation amongst x and y is 0, then x = y)
2. $d(x, y) = d(y, x)$ (The separation from x to y is the same as the separation from y to x)
3. $d(x, y) + d(y, z) >= d(x, z)$

The three criteria's specified above are known as Triangle Inequality. The Triangle Inequality expresses that the way from x to z must be no longer than any way that experiences another middle of the road point (the way from x to y to z).for case we can't attract a triangle which it's speedier to achieve a point from going along two sides as opposed to coming to the opposite side.

These three criteria, essential as they seem to be, are all that is required for something, for instance, the Levenshtein Distance to qualify as a Metric Space. In arithmetic, a metric space is a set for which separates between all individuals from the set are characterized. Presently take two components, the question component which we are utilizing for the hunt and n the greatest separation a component can have from the inquiry component[4]. How about we take an irregular component from the settled set and think about them two, expect that we get the Levenshtein Distance as d. Presently as the Triangle imbalance holds every one of our outcomes must lie between separations d - n to d + n. Presently we have all the comprehension to make a BK-Tree.

II. OPERATIONS

BK tree supports two operations i.e. 1)Create and 2)Search

2.1 Create operation

Suppose we have the dictionary data as {“BALL”,”WALL”,”TAIL”}

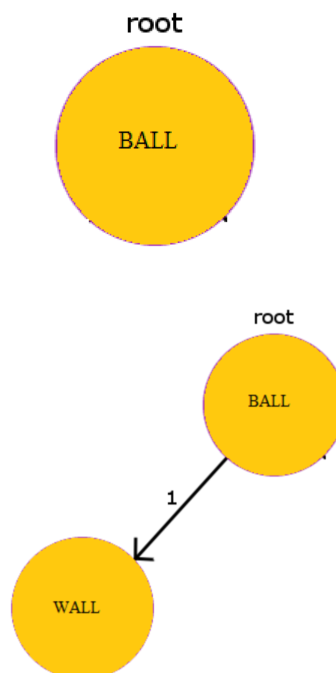
The nodes in the BK-Tree will show the elements in our dictionary and there will be exactly the same number of elements as the number of words in our dictionary given.

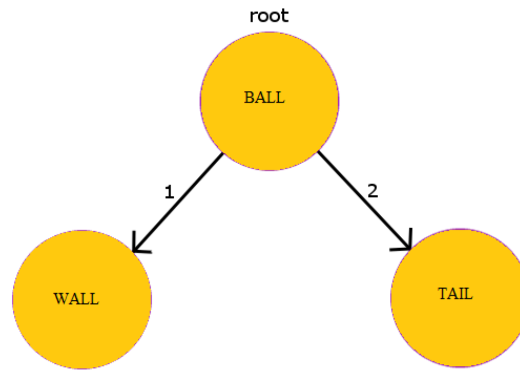
Here for given dictionary, it is n=3(three nodes).The edges between nodes show the edit distance(Levenshtein Distance d).The first element is the root, then we take the Levenshtein Distance d from the root and add the next elements on the tree like this:

LevenshteinDistance(BALL, ***WALL***) -> 1

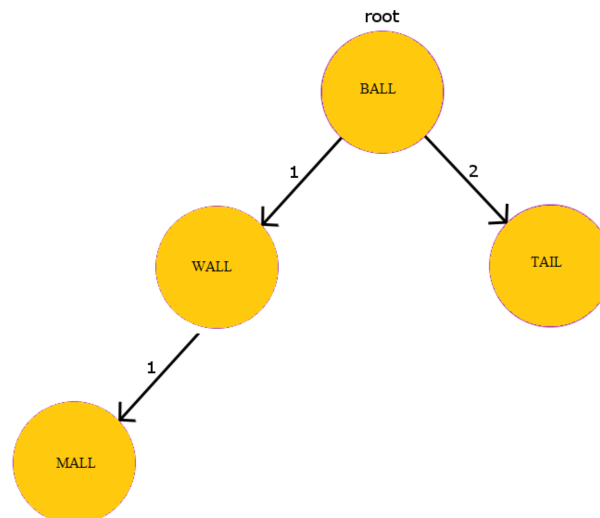
LevenshteinDistance(BALL, ***TAIL***) -> 2

The value of d between BALL and WALL is 1 and d is 2 for BALL and TAIL.



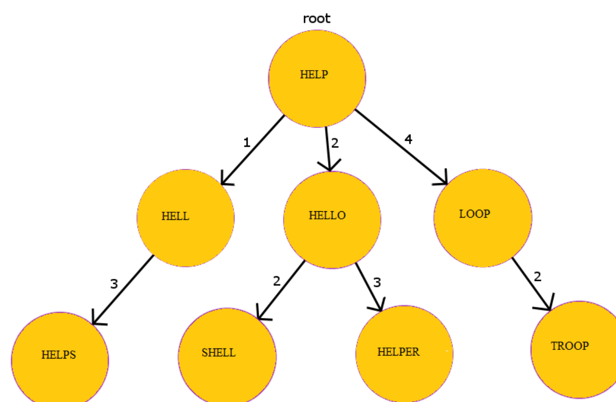


Here the tree is created and each node will have only one child with same edit distance. If a new word "MALL" is added then it cannot be added as a child to the root as it has already had child with d=1.It is added to the node "WALL" as it has d=1.



2.2 Search operation

Now to find the nearest correct word or the string matching here is the example of BK tree using the given dictionary:



The simple method to find a word is start from the root and move to left and right till the edit distance is the minimum till the end.

To find the corrected or misspelled word we have to define the terms i.e. tolerance value.This **tolerance value(T)** is highest edit distance from our misspelled word to the correct words in our dictionary.

BK tree is constructed based on edit distance calculated and searching for misspelled word can be found out using by searching over children with edit distance [d-T] to [d+T].

Suppose we have an incorrectly spelled word "oop" and T is 2. Presently, we will perceive how we will gather the normal right for the given incorrectly spelled word.

Step 1: We will begin checking the value d from the root hub. d("oop" - > "help") = 3. Presently we will emphasize over its children having in range [d-T,d+T] i.e [1,5]

Step 2: Let's begin emphasizing from the most noteworthy value d i.e node "loop" with d=4 .Now at the end we will discover its distance from our incorrectly spelled word. d("oop","loop") = 1.

here d = 1 i.e d <= T , so we will include "loop" to the normal right word rundown and process its child elements having alter remove in range [d-T,d+T] i.e. [1,3]

Step 3: Now, we are at position "troop". Finally, we will check its distance from the incorrectly spelled word. d("oop","troop")=2. Here again d <= T, thus again we will include "troop" to the normal right word list.

We will continue the same for every one of the words in the range [d-T,d+T] beginning from the root position till the base most leaf node.

In this manner, toward the end, we will be left with just 2 expected words for the incorrectly spelled word "oop" i.e {"loop","troop"}

III. ANALYSIS

The basic method to find the nearest match is take all word in the dictionary and compare the edit distance(d) with tolerance value (T) this will take huge amount of time i.e. O(N1*M*N2)

where N1 is a number of words , N2 is the length of incorrect word and M is mean size of the perfect match.

But by using a BK tree we can reduce this time complexity in the following manner; assuming **tolerance limit(T)** to be **2**. Now approximately, the depth of BK-Tree will be log N, where N number of elements. At every level, we are visiting 2 elements in the BK tree and doing edit distance evaluation. Therefore, our Time Complexity will be **O(N1*N2*log N)**, here **N1** is the mean length of the string in our dictionary and **N2** is the length of the incorrect word.

IV.APPLICATION

BK tree usually used in the spell checking applications like in dictionary, text editors where we write spelling wrong help in correcting the word as it is relatively simple it has mainly three parts firstly it checks whether the word exists in the dictionary or not, secondly find the possible fixes for misspelled word and lastly order suggestions based on some sort of heuristic, it takes linear time by scanning all words in the dictionary and calculating edit distance, it is really an amazing data structure for building a dictionary of similar words and it also used to guess the typed word like "cat" when we wrote "cta" it works with the words from dictionary with the help of the first word which act as a root node then with the help of the Levenshtein distance subsequent words are attached. And also used in the string matching applications and various soft wares were correct

features are a prerequisite for auto-correcting the word. It has wide application in search engines for many websites for correcting the spelling for naïve users.

V. FUTURE SCOPE

As it is a data structure for performing to spell check based on edit distance that is Levenshtein concept and it is being used in the dictionary for easier searching of the words or checking the misspelled word and correcting it with right ones it has auto-correcting features so somewhere it will be prerequisite for the future generations soft wares where there is need of the auto-correction tool so it will be demanded by such circumstances and useful for the inbuilt functionality of the soft wares where the wrong typed words automatically changes to the correct ones without clicking on the wrong ones , this could be better improvement over the spell checking applications and help in avoiding the misspelled words and correcting them automatically.

VI. REFERENCES

[1] http://blog.not.net/2007/4/Damn-Cool-Algorithms-BK-Trees

[2] https://dzone.com /algorithm-week-bk-trees-part-1

[3] http://www.geeksfgeeks.org/BK-tree-introduction /

[4] http://blog.mishkokyi.net/posts/2015/Jul/31/implementing-bk-tree

[5] https://nulwords.wordpress.com/2013/03/13/the-bk-tree -spell-checking/