BK Trees : Efficient Spell Checking in NLP

November 5, 2023

Hemlata Gautam (2022CSB1084) ,
Neha Dahire (2022CSB1096) ,
Smriti Gupta (2022CSB1128)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
E Harshith Kumar Yadav

Summary: We would like to access the words in the dictionary that are very close to the misspelled word as fast as possible and to recommend them to the client. BK-trees are data structures that will help us improve our recommendation speed to another level compared to the naive linear approach. It uses string searching based on Levenshtein distance (edit distance). They form a tree structure, with each node representing a word and its children representing words within a specific Levenshtein distance from the parent word. To build a BK-Tree, words are inserted one by one, starting with an empty tree, and compared based on their edit distances. During a search operation, one starts at the root node and traverses the tree, considering the Levenshtein distances between the target word and the words associated with each node. If the distance exceeds a predetermined threshold, the search terminates as no closer matches can be found. BK-Trees are commonly applied in spell-checking, auto-correction, and approximate string matching scenarios, where they provide an efficient way to find words or strings similar to a given input.
1)construction:
Insertion takes O(h), where h is the height of the tree. In the worst case, the tree can become unbalanced, resulting in O(n) time complexity, where n is the number of words in the tree.
2)Approximate string matching: Finding words within a given Levenshtein distance threshold: O(d * log(n)), where d is the Levenshtein distance threshold and n is the number of words in the tree. This assumes a balanced tree.

---

1. Introduction

Bk-trees, also known as Burkhard-Keller trees, are a type of tree data structure primarily used for quickly finding the closest matches to a given input string. They were first proposed by Walter A. Burkhard and Robert M. Keller in 1973.
The main application of bk-trees is in the domain of approximate string matching, where the goal is to find strings that are similar to a given query string. This makes them particularly useful in fields like spell checking, data compression, and computational biology.
The structure of a bk-tree is based on the concept of a metric space, where a distance metric, such as the Levenshtein distance or Hamming distance, is defined between elements. Each node in the tree represents a string, and the children of a node are determined by the distance of their associated strings to the parent node's string, based on the chosen distance metric.

2. Equations

In a BK-tree, the essential equation revolves around the distance metric used to measure the dissimilarity or similarity between elements. For instance, if you are using the Levenshtein distance for string matching, the

equation to compute the distance between two strings s and t can be expressed as follows:
d(s,t)= lev(|s|, |t|)
where d(s,t) is the Levenshtein distance between strings s and t, and lev(i,j) is the recursive function that computes the Levenshtein distance between prefixesw of s and t.
This equation demonstrates the fundamental mathematical calculations that are utilized in the process of building and searching BK-trees. However, the focus of BK-trees is not on this equation itself but on the application of these distance metrics to efficiently organize and retrieve similar strings.

3. Figures and Algorithms

3.1. Figures

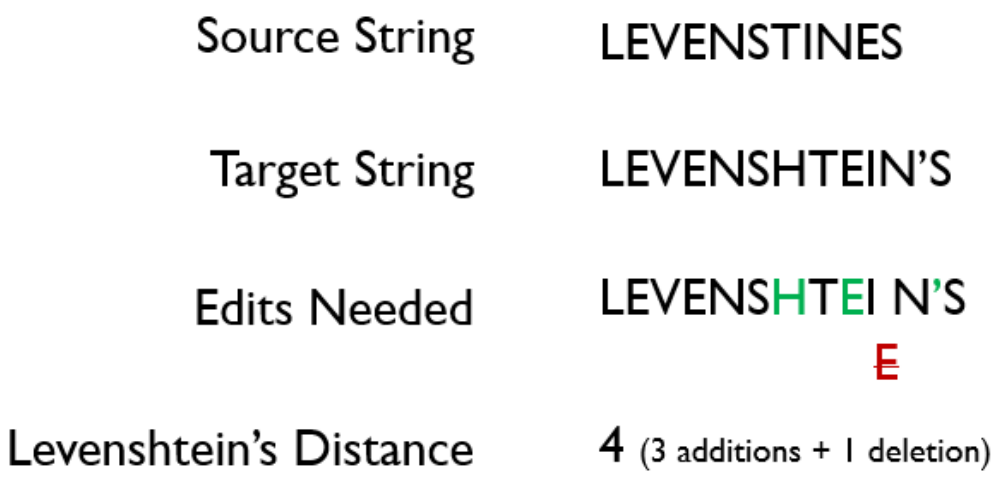

Figure 1: Calculation of Levenshtein distance

Figure 1 clearly explains the procedure to find Levenshtein distance between two strings. To convert the source string into target string we have to make 3 additions 'H', 'E', "' and 1 deletion of 'E'. Therefore, total 4 operations were needed to convert the source string into the target string. Hence, the Levenshtein distance is 4.

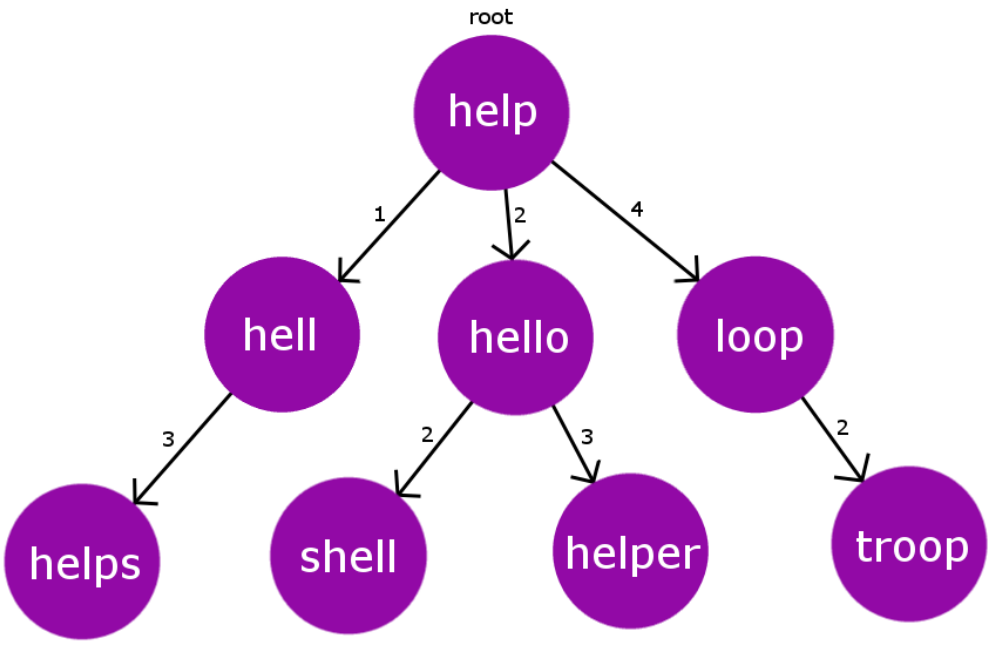

Figure 2: The structure of a BK-Tree

Figure 2 demonstrates the structure of a bk tree. It has a root and then children at Levenshtein distance away which is displayed as edge weight.

3.2. Algorithms

A BK-Tree is constructed by inserting words into an initially empty tree. Each insertion starts at the root node and progresses down the tree by determining the Levenshtein distance between the word being inserted and the word associated with the current node. If the distance is zero, the word is already in the tree, and no further action is taken. If the distance is greater than zero, the word is added as a child of the current node.

4. Algorithm for BK-Tree Construction

The insertion primitive is used to populate a BK-tree t according to discrete metric d.

1. for each word W in the dataset:
2. if t is empty:
3. if t has no root node:
4. create a node r in t
5. $W_r \leftarrow W$
6. Return r
7. end if
8. end for
9. while there are more words to insert into the tree:
10. Set u to the root of t
11. end while

The BK-tree insertion operation for discrete metric d is now complete.

5. Some further useful suggestions

Proposition 5.1. *Edit Distance: Given two strings str1 and str2 of length M and N respectively and below operations can be performed on str1.*

- Operation 1 (INSERT): Insert any character before or after any index of str1.
- Operation 2 (REMOVE): Remove a character of str1.
- Operation 3 (REPLACE): Replace a character at any index of str1 with some other character.
  The minimum number minimum number of edits (operations) to convert 'str1' into 'str2' is the editDistance.

6. Conclusions

Burkhard-Keller Trees (BK-trees) are hierarchical data structures used for efficient approximate string matching. They enable quick searches for words within a specified edit distance, making them valuable for tasks like spell-checking. BK-trees maintain a balanced structure, are adaptable for dynamic updates, and can be used beyond strings. However, they are most effective when the edit distance metric is utilized and may not be optimal for all similarity search scenarios.

7. Bibliography and citations

BK-Tree Wikipedia Article: The Wikipedia article titled "BK-TREE" [1] serves as a comprehensive resource for understanding the BK-Tree data structure. It provides valuable insights into its definition, properties, and various applications in spell checking and auto correct.
Nitish Kumar's article on Geeks for Geeks titled "BK-TREE" [2] provides an introduction to the BK-Tree along with insights into its implementation. It serves as a valuable resource for those interested in the practical aspects of BK-Trees, including code examples and demonstrations.
Shridhar Shah's article "SPELL CHECKER AND STRING MATCHING USING BK-TREES" [3] delves into the practical applications of BK-Trees in the domain of spell checking and string matching. Published in 2017,

this article offers a deeper understanding of how BK-Trees are employed in real-world scenarios, particularly in text processing and correction tasks.

Acknowledgements

"We would like to express our heartfelt gratitude to our esteemed instructor, Dr. Anil Shukla, for providing us with the invaluable opportunity to delve into the intricacies of this vital data structure. Additionally, we extend our sincere thanks to our dedicated project supervisor, Mr. E. Harshith, for his unwavering guidance and support throughout this endeavor. Our academic institution played a pivotal role by providing essential resources and fostering an environment conducive to learning.
We are also deeply appreciative of the unwavering support from the open-source community, which proved to be indispensable to the success of our BK-tree project. We extend our sincere thanks to all individuals and that contributed to our project."

References

[1] Bk-tree,https://en.wikipedia.org/wiki/bk-tree.

[2] NITISH KUMAR. Bk-tree,https://www.geeksforgeeks.org/bk-tree-introduction-implementation/.

[3] Shridhar Shah. Spell checker and string matching using bk-trees. pages 343–346, 2017.

8. Appendix: Additional Resources and References

In this appendix, we provide a collection of valuable resources and references that will aid in understanding and implementing BK Trees, a powerful data structure for approximate string matching and spell checking. These resources cover key concepts, practical implementations, use cases, and additional insights into the world of BK Trees.
1. Levenshtein Distance Algorithm: Navarro, G. (2001). A guided tour to approximate string matching. ACM Computing Surveys (CSUR), 33(1), 31-88. Provides detailed insights into the Levenshtein distance algorithm, which is a critical component of BK Trees.
2. GitHub Repository: Example BK Tree implementation in Java: `https://github.com/kiababashahi/Spell-check-and-BK-Trees/blob/master/README.md` The project referenced in the main text includes a simple BK Tree implementation. It can serve as a starting point for practical implementation.
3. Use Cases and Applications:
- Navarro, G.,and Raffinot, M .(2002). Flexible pattern matching in strings. - Practical On-line Search Algorithms for Texts and Biological Sequences.
-Cambridge University Press. Discusses applications of BK Trees and other string matching algorithms in various fields, including bioinformatics.
4. Spell Checking and Auto-correct: - Norvig, P. (2007). How to Write a Spelling Corrector: `http://norvig.com/spell-correct.html` - An insightful article by Peter Norvig, which delves into spelling correction and auto-correction, explaining related algorithms.