Automatic Spell Checker using Damerau-Levenshtein Distance Algorithm

Sarah Blankenship - smblank@clemson.edu

November 29, 2021

Abstract

While most automatic spell checkers available rely on a machine learning approach, implementing these can be very complicated and takes much experience to do well. So this project is focused on creating a spell checker that navigates a String B-Tree and finds alternate spellings for words using the Damerau-Levenshtein edit distance. The analysis of this implementation showed that this approach could maintain the same speeds that would be expected but needed more work to meet the accuracy needed.

1 Problem Formulation

Most word processing software available comes with an automatic spell checker. However, most of these spell checkers rely on machine learning models that can continuously learn the user, so this project attempted to create a spell checker based on efficient data structures. This spell checker was created using the Damerau-Levenshtein edit distance and a String B-tree to store the dictionary of accepted words for the user. The spell checker can navigate the String B-Tree to check for correctly spelled words, and if not, find words within the dictionary with a small Damerau-Levenshtein edit distance.

1.1 Problem Background

1.1.1 Damerau-Levenshtein Automata

A *Levenshtein Automata* is a finite state machine that can recognize all strings within a given edit distance of a given string. This automaton defines the edit distance as the number of characters that need to be inserted, deleted, or substituted to transform one string into another [6]. This edit distance helps implement a spell checker, as a Levenshtein Automata can find the closest strings to the given one by choosing strings within a certain edit distance. A variant known as the *Damerau-Levenshtein* approach exists to improve accuracy. This approach is very similar to the Levenshtein Automata, but it also includes transpositions as an accepted edit [11]. This inclusion would result in more strings

fitting within a small edit distance, resulting in a higher accuracy of the spell checker. It accounts for the general spelling error of mixing up two letters. The Damerau-Levenshtein approach has a slightly longer processing time, but many researchers have proposed optimizations to make up for it. One such study proposes splitting up the dictionary of accepted strings into multiple dictionaries based on the length of string, which showed improved speed [10].

1.1.2 String B-Tree

The Levenshtein Automata (even using the Damerau-Levenshtein approach) can be implemented using a String B-Tree [6]. A String B-Tree is a combination of a regular B-Tree and a Patricia Trie. Each node of a String B-Tree contains a Patricia Trie. This approach allows the String B-Tree to make use of Patricia Tries' efficiency. A Patricia Trie combines words that are spelled alike and branches on letter differences [8]. This structure makes for a very efficient search algorithm as entire subtrees in a Patricia Trie can be ignored. The B$^+$-Tree organizes these individual Patricia Tries. A B$^+$-Tree keeps the same balance of a regular B-Tree, but only keeps data pointers at leaf nodes which decreases search time and makes insertion and deletion of nodes easier.

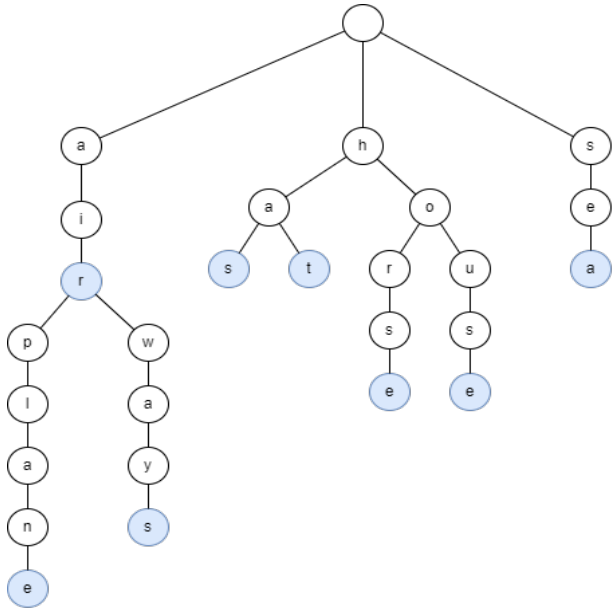

Figure 1: Patricia Trie structure for a small dictionary. The top empty node is the root node, and shaded nodes represent complete words.

B$^+$-Trees can work with a large amount of data, but not when the nodes' keys are long, while Patricia Tries work well with individual strings of any length,

but not with an extensive data set [4]. By organizing the data and pointers to corresponding pages of memory, this combination results in a data structure that can outperform a normal B-Tree in both speed and disk accesses, primarily when focused on search operations like in this project [3].

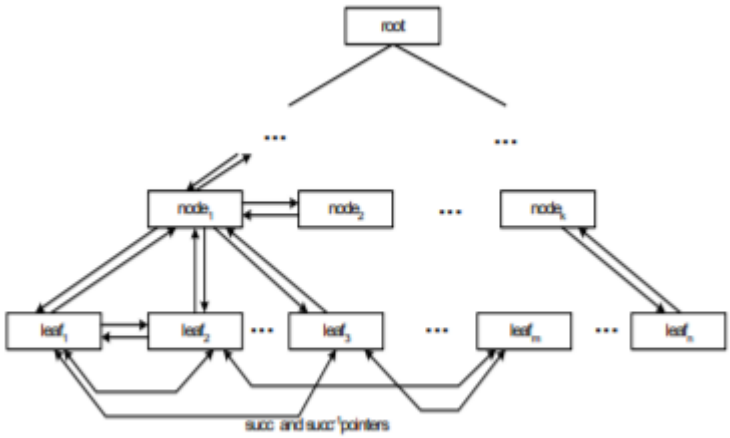

Figure 2: String B-Tree Structure [3]

1.2 Scope & Deliverables of Project

This project focuses on implementing a String B-Tree data structure and designing a Damerau-Levenshtein Automata that can navigate said tree and return a list of words with a given edit distance based on a user inputted word. The user can view the feedback from the automata, including marking misspelled words and viewing the alternative spelling suggestions for that word. Other sections explore the possibility of adapting this program for other purposes, but this project will not be implementing those adaptations.

This project will deliver a complete spell checker program on completion. For those using the complete program, there will be a GUI to allow for text editing that also marks the incorrectly spelled words, lists suggestions, and allows the user to choose a replacement word. The backend of this program will include an implementation of a String B-Tree (and by extension Patricia Trie) in Java and an algorithm to retrieve Damerau-Levenshtein edit distances and navigate the String B-Tree to return a list of words within a certain Damerau-Levenshtein edit distance. This implementation was kept separate using software development best practices so that the algorithm could be adapted to other uses.

1.3 Problem Through Lens of Data Structures

This project is interesting as a data structures problem as it combines different data structures to create a more efficient structure in searching and memory. The average case time complexity for a B^+-tree for a lookup is $O(T \log_T N)$ [9]. For a Patricia Trie lookup, the average case time complexity is O(K) and the worst-case time complexity is O(D) [8]. So for the combination of a B^+-tree and a Patricia Trie to form a String B-Tree, its average case search time complexity would be $O(T \log_T T + K)$. And the space complexity of a String B-Tree is $O(1 + W / P + \log_P N)$, where W is the length of the word being checked, and P is the size of a page of memory [1].
The completed software will be a functional spell checker, which is a general use tool. Part of this project is to see if this implementation is comparable to the commercial spell checkers available. The Damerau-Levenshtein edit distance can also be used with a machine learning approach to spell checking in order to fine-tune its results [5]. This approach can also be adapted to more specialized areas such as data analysis or DNA sequencing [11].

2 Strategy

The strategy for this project is first to use a dictionary of common English words and organize them into a String B-Tree according to their relative Damerau-Levenshtein edit distances. Once the tree is built, an algorithm can be developed around it to find strings that match a user-inputted string with spelling errors. The closest matches are found by traversing through the tree and finding the words within the dictionary with the closest Damerau-Levenshtein edit distance. As stated above, The String B-Tree is a combination of the B^+-Tree and Patricia Trie data structures [4]. The main algorithm of this program is implementing the Damerau-Levenshtein finite-state automata by navigating the String B-Tree. The node of each String B-Tree contains a Patricia Trie, which was implemented as a standalone object. The dictionary was split amongst pages of memory to cut down on disk accesses, and each Patricia Trie is built from the words contained on one page of memory. The Patricia Trie is built by maintaining a parallel ArrayList to the memory page to track which character is the next "node" in the trie.

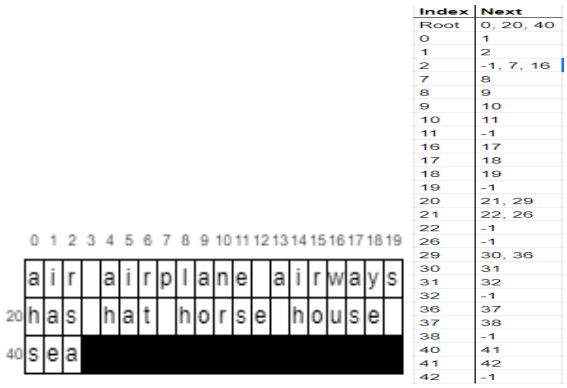

Figure 3: Representation of parallel arrays to maintain Patricia Trie

The Patricia Trie object also contains pointers to other nodes within String B-Tree, specifically pointers to the parent, previous and next nodes at the same depth, and any children. The B^+-Tree aspect was implemented by adapting Java Swing's tree library. The DefaultMutableTreeNode class implements the B^+-Tree, despite its name. This class can accept any object to store in the node and keeps track of all node children [7]. This class was extended to maintain the balance of a B^+-Tree and update the double pointers stored in the Patricia Trie object.

The Damerau-Levenshtein distance algorithm will take a user-inputted string and traverse through the string B-Tree to find words within a set edit distance. Suppose the word does not exist within the dictionary. In that case, the algorithm will go through the Patricia Trie to get a list of words in the dictionary but within a given edit distance (defined as a constant variable in the program itself). This algorithm goes into each branch of the Patricia Trie until reaching a point where that branch is outside the defined edit distance. Then the algorithm pulls back for that branch and repeats until finishing the process. This process allows the program to prune entire subtrees that contain words utterly different from the input, which speeds up execution time.

The software developed will have a simple GUI for users to enter text and quickly receive the program's feedback. This project was already using the Java Swing's tree library to implement the String B-Tree, but Java Swing's primary purpose is implementing GUI's. The base UI will be a text editor to enter and edit the text as they see fit. These words will be fed to the Damerau-Levenshtein algorithm to determine if they are correctly spelled, and if not, they will be marked wrong and have a list of alternative spellings provided to the user. Finally, the user can select an alternative spelling to replace their spelling in the text.

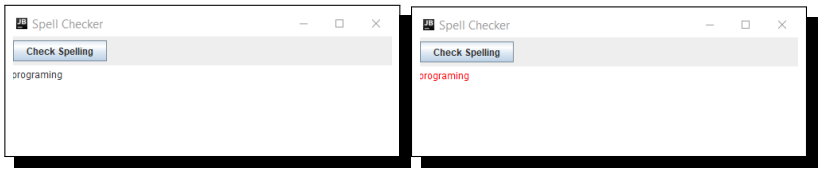

Figure 4: Spell checker GUI showing a word being marked incorrect

3 Analysis

3.1 Methodology

The analysis of this project was conducted through two tests. The first test was a real-time experience of the spell checker using the UI. This test was less precise, but the qualitative results were also valuable. This qualitative data reflected the user experience and how quick it was in general use. The other test collected more concrete qualitative data by calling the spell checker function directly with a specific word and recording the replacement words returned and the runtime. This data gave a much more concrete evaluation of the spell checker.
The spell checker was evaluated on two metrics: speed and accuracy. Speed is a significant concern for most programs, and a spell checker needs to be fast enough to keep up with a consistently changing input. This project is also interested in seeing if this implementation of a spell checker can keep up with the available commercial spell checkers. The second metric is equally essential to a spell checker, as it must accurately return a list of alternative suggestions for a misspelled word. Since most commercial spell checkers rely on a machine learning model that learns the user, it was interesting to see the accuracy of the lists the spell checker returns and how often the intended word was found.

3.2 Results

The speed test showed the expected results of using a String B-Tree to implement this project. The ability to avoid entire branches of the Patricia Tries made searching each node a lot quicker, and the organization of them into a B^+-Tree helps for searching an extensive dictionary. The UI tests used several common typing errors, and the UI remained responsive and able to keep up with changes to the input. The scripted tests used common misspellings, especially errors that are likely to happen while typing. These tests included a few words not included in the dictionary to test their speed. As shown in , most of the execution times were very close to zero milliseconds, with only one outlier.

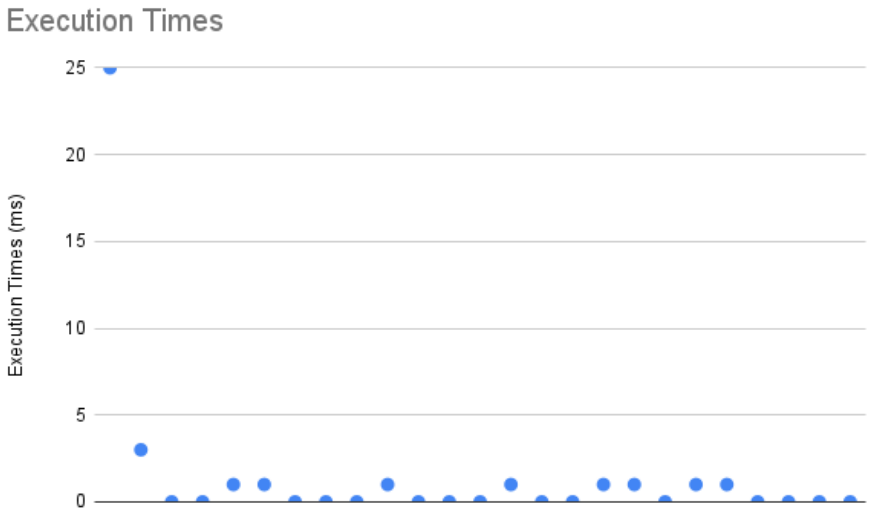

Figure 5

The accuracy test had more surprising results than the speed test. Many of the tests did not return the intended word as a part of the alternative list. This list also often included the correct word further down in the list, rather than being closer to the top, which would be preferable. However, the program could still identify if a word was misspelled almost every time, both in scripted and UI tests.

3.3 Discussion

The speed test gave some promising results for this project. The execution time is what was expected from starting this project. More work is needed to refine the Damerau-Levenshtein algorithm. There seem to be issues navigating the entire tree and refining the list of alternative words to present to the user. The foundation is already there to implement these improvements.

4 Future Work

More work is needed to fine-tune this spell checker to be on par with the commercial spell checkers available. The biggest issue is that the list of alternate words needs to be refined and reorganized based on edit distances before being presented to the user, rather than solely relying on the structure of the Patricia Trie. The obvious next step for this project would be adding the functionality necessary for working with other languages. For many languages that share the same alphabet, this will be a simple task. However, some other languages with

more complex alphabets or other written systems will present a challenge to this implementation or may not even work.
Another improvement to the system would be to add a cache system for common misspellings. This improvement would be helpful to increase speed, particularly if the size of the dictionary increases. A cache would also add an element to the program to learn the misspellings the user does the most often and can pull those up quicker or potentially auto-correct the misspelling.

5 Conclusion

This project aimed to create an automatic spell checker without using machine learning models. There is much research on using a Patricia Trie or String B-Tree because the data structure makes searching for a word based on slight spelling differences quicker. Adding a Damerau-Levenshtein edit distance to this can expand the search for the correct word. This expanded search theoretically would result in higher accuracy but needs more refinement in practice to achieve this. This refinement to accuracy is needed for this spell checker to be comparable to a spell checker's traditional machine learning implementation.

References

[1] Michael A Bender, Martin Farach-Colton, and Bradley C Kuszmaul. "Cache-oblivious string B-trees". In: *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 2006, pp. 233–242.

[2] Sarah Blankenship. *spellChecker*. Last accessed 29 September 2021. 2021. URL: `https://github.com/smblank/spellChecker`.

[3] X. Fan, Yu Yang, and L. Zhang. "Implementation and Evaluation of String B-Tree". In: 2004.

[4] Paolo Ferragina and Roberto Grossi. "The String B-Tree: A New Data Structure for String Search in External Memory and Its Applications". In: *J. ACM* 46.2 (Mar. 1999), pp. 236–280. ISSN: 0004-5411. DOI: `10.1145/301970.301973`. URL: `https://doi.org/10.1145/301970.301973`.

[5] Kristin H. Huseby. *How to improve the performance of a machine learning model with post processing employing Levenshtein distance*. Last accessed 8 September 2021. 2020. URL: `https://towardsdatascience.com/how-to-improve-the-performance-of-a-machine-learning-model-with-post-processing-employing-b8559d2d670a`.

[6] Jules Jacobs. *Levnshtein automata can be simple and fast*. Last accessed 6 September 2021. 2015. URL: `https://julesjacobs.com/2015/06/17/disqus-levenshtein-simple-and-fast.html`.

[7] Oracle. *Class DefaultMutableTreeNode*. Accessed 11 Nov. 2021. URL: https://docs.oracle.com/javase/6/docs/api/javax/swing/tree/DefaultMutableTreeNode.html.

[8] Bruno Osiek. *Comparing PATRICIA Trie, With It's Nuts and Bolts, With Hash Map*. Accessed 11 Nov. 2021. Feb. 2020. URL: https://medium.com/@brunoosiek/patricia-tries-nuts-and-bolts-170a317b3ab6.

[9] Kerttu Pollari-Malmi. *B+-Trees*. Accessed 11 Nov. 2021. URL: https://www.cs.helsinki.fi/u/mluukkai/tirak2010/B-tree.pdf.

[10] Utomo Pujianto, Aji Prasetya Wibawa, and Raditha Ulfah. "Dictionary Distribution Based on Number of Characters for Damerau-Levenshtein Distance Spell Checker Optimization". In: *2020 6th International Conference on Science in Information Technology (ICSITech)*. 2020, pp. 180–183. DOI: 10.1109/ICSITech49800.2020.9392059.

[11] C. Zhao and S. Sahni. "String correction using the Damerau-Levenshtein distance". In: *BMC Bioinformatics* 20.277 (2019).